Jørn Lorentzen og Henrik Nellager

Maskinkode med Amstrad

Borgen

Maskinkode med Amstrad

Jørn Lorentzen og
Henrik Nellager

Maskinkode med Amstrad

Borgen

Copyright © 1985 Jørn Lorentzen og
Henrik Nellager
Omslag: Henrik Koitzsch
Trykt hos Narayana Press
ISBN 87-418-7533-8

Alle rettigheder forbeholdes,
herunder retten til erhvervsmæssig
udnyttelse af bogens programmer.

Indhold

Forord *9*

Kapitel 1 Introduktion *11*

Vi gennemgår fordele og ulemper ved maskinkode, forklarer kort om Amstrad's opbygning og bringer et demonstrationsprogram.

Kapitel 2 LD-instruktionen, del 1 *17*

Vi lærer processorens registre at kende, samt hvordan man giver dem værdier.

Kapitel 3 LD-instruktionen, del 2 *23*

Vi lærer at læse fra og ændre i adresseringsområdet og lærer flere registre at kende.

Kapitel 4 Bit-instruktioner *36*

Vi omtaler de logiske instruktioner, AND, OR og XOR, og RES, SET og BIT, der opererer på de enkelte bits.

Kapitel 5 Simpel aritmetik *43*

Vi lærer at addere med INC, ADD og ADC og at subtrahere med DEC, SUB og SBC.

Kapitel 6 Hop-instruktioner *52*

Vi hopper frem og tilbage i programmer ved hjælp af instruktionerne JP, JR og DJNZ.

Kapitel 7 Sammenligning *58*

Vi sammenligner registre ved hjælp af CP samt lærer brugen af bloksøge-instruktionerne CPI, CPIR, CPD og CPDR. Desuden nævner vi små, nyttige instruktioner.

Kapitel 8 Stakken *65*

Vi lærer at gemme værdier med PUSH og hente dem tilbage med POP. Endvidere omtaler vi andre instruktioner opererende på stakken.

Kapitel 9 Subrutiner *74*

Vi kalder underrutiner med CALL og returnerer med RET. Vi lærer desuden at bruge operativsystemets rutiner.

Kapitel 10 Flere blokinstruktioner *82*

Vi lærer at flytte hele blokke af data med instruktionerne LDI, LDIR, LDD og LDDR.

Kapitel 11 Rotationsinstruktioner *86*

Vi lærer at flytte rundt på de enkelte bits i registrene og briefer kort om BCD-aritmetik.

Kapitel 12 I/O-portinstruktioner *98*

Vi kigger på instruktioner, som sender data ud og ind af portene.

Kapitel 13 Adresseringsområdets opbygning *103*

Vi gør rede for opbygningen af ROM og RAM og gennemgår RST-instruktionerne.

Kapitel 14 Tastaturet *113*

Vi lærer at aflæse tastaturet ved hjælp af operativsystemet.

Kapitel 15 Tekstskærmen *120*

Vi ser på tekstskærmens opbygning og lærer at håndtere f.eks. vinduer, pen og baggrund.

Kapitel 16 Grafikskærmen *134*

Vi lærer at udnytte Amstrad's grafiske muligheder, såsom plotning af tegning af linjer.

Kapitel 17 Lyd *156*

Vi belyser Amstrad's audiovisuelle egenskaber.

Kapitel 18 Interrupts/events *163*

Vi lærer at afbryde programmer, uanset hvad disse udfører, for at eksekvere vigtige instruktioner. Vi lærer desuden at kende visse instruktioner til brug ved interrupts.

Kapitel 19 De alternative registre *172*

Vi ser på, hvordan man kan benytte de alternative registre – dog med lidt besvær.

Kapitel 20 Gode råd *179*

Vi giver maskinkodeprogrammøren gode råd med på vejen og bringer en oversigt over samtlige instruktioners virkemåde.

Kapitel 21 Firmware-rutiner *187*

Vi ser på rutinerne i operativsystemet og forklarer, hvordan de kan bruges.

Kapitel 22 Ur *278*

Vi bringer et ur-program.

Kapitel 23 Ekstra kommandoer *296*

Vi ser på, hvordan vi kan udbygge Amstrad's BASIC ved hjælp af ekstra, selvdefinerede kommandoer.

Kapitel 24 Singlestep *310*

Vi bringer et stort program til brug ved fejlfinding. Programmet er i alt på 1,2 K.

Appendices:

Appendiks A Instruktionernes hex-koder *331*
Appendiks B Flagsætning *343*
Appendiks C Konvertering mellem talsystemer *347*
Appendiks D T-states *350*
Appendiks E Kontrolkoder *356*

Større programmer i kapitlerne 1-19:

Kapitel 1 Demonstration *14*
Kapitel 9 Udskrivning af decimale tal *80*
Kapitel 10 Lagring af skærmbillede *84*
Kapitel 11 Udskrivning af binære tal *90*
Kapitel 11 Udskrivning af hexadecimale tal *96*
Kapitel 15 Forstørret gengivelse af karaktersættet *132*
Kapitel 16 Flag *136*
Kapitel 16 Søjlediagram *141*
Kapitel 16 Pixelscroll *153*
Kapitel 17 Vuffelivov *160*

Forord

Det er vort formål med denne bog at få udbredt kendskabet til det fascinerende programmeringssprog maskinkode, der er datamatens eget sprog. Bogen er skrevet til hjemmedatamaten Amstrad CPC 464 og kan bruges fuldt ud på Amstrad CPC 664 og CPC 6128, men sproget benyttes af alle computere, der fungerer på basis af en Z80-processor.

Det er en fordel for dig, hvis du på nuværende tidspunkt er fortrolig med Amstrad og dens muligheder igennem dens BASIC-version. Du vil dog alligevel, tror vi, undervejs blive forbavset og måske endda foruroliget over den kapacitet, som Amstrad i virkeligheden har – men som ikke kan benyttes i BASIC. Det kan den derimod i maskinkode!

Vi starter indlæringskurset helt fra bunden med de simpleste ordrer. Vi har ved udarbejdelsen lagt vægt på, at der undervejs er mange forklarende eksempler, da vi mener, at den praktiske udførelse af den teoretiske viden er kernen til indlæringen. Efterhånden som bogen skrider frem, forøges størrelsen af programmerne i takt med stigningen i sværhedsgrad. Eksemplerne er valgt med omhu for at give den bedste indlæringsmulighed, og alt er forklaret undervejs. Alt dette skal forhåbentlig medføre, at du, efter at have læst denne bog og fået lidt praktisk erfaring, er i stand til at udvikle dine egne maskinkodeprogrammer.

Indholdsfortegnelsen er den naturligste kilde til at finde ud af bogens opbygning kapitel for kapitel, og dér kan du også se, hvilke større programeksempler der er i bogen. Kapitel 1 er en slags introduktion, der bl.a. indeholder et stort demonstrationsprogram. Kapitlerne 2-12 giver den fornødne teoretiske baggrundsviden til maskinkodeprogrammering, og disse kapitler er de mest generelle, da de primært er baseret på Z80-processoren og ikke på Amstrads egenskaber. Kapitlerne 13-19 tager derimod direkte fat i CPC'ens muligheder igennem bl.a. grafik, farver og lyd. Ba-

gest er, foruden diverse appendices og et utroligt stort kapitel fungerende som opslagsværk, samlet 3 større programmer.

Vi vil råde dig til under indlæringen at sidde i nærheden af Amstrad-computeren, da du – for din egen skyld – *skal* indtaste alle eksempler. Det vil give den bedst mulige forståelse.

Vi takker alle, der har været behjælpelige under udarbejdelsen og ønsker dig hermed god fornøjelse med maskinkodeprogrammeringen.

Jørn Lorentzen & Henrik Nellager
Avedøre, april 1985

Kapitel 1

Du har sikkert allerede nu brugt mange timer på at programmere din Amstrad CPC i BASIC. Nu står du over for indledningen til en ny æra indenfor programmering. I stedet for at skulle arbejde med det langsomme BASIC, vil vi nu vejlede dig i maskinkode, så du kan lære at bruge datamaten til dens yderste grænse.

Maskinkode har flere ulemper. For det første er alle de behagelige kommandoer og funktioner, som du kender fra BASIC, forsvundet. Når man arbejder i maskinkode, kobler man den indbyggede BASIC-fortolker fra. Det er den, der gør, at du kan programmere i BASIC og ikke er nødt til at bruge maskinkode, hvis du ikke har lyst. Uden alle BASIC's kommandoer vil maskinkoden i starten forekomme besværlig og uoverskuelig, men du vil hurtigt glide ind i maskinkodeprogrammørens rytme og lære at arbejde med sproget. Det består faktisk kun af ettaller og nuller, der tilsammen udgør det, vi kalder det binære talsystem. Grunden, til at en computer arbejder på dette noget primitive niveau, er, at den kun kan skelne mellem høj og lav spænding, angivet ved 1 og 0.

En anden ulempe er, at der ikke forekommer fejlrapporter. Når der i et maskinkodeprogram er en fejl, så vil det i mange tilfælde have samme effekt som et reset af computeren. Selv hvis dette ikke er tilfældet, så forekommer der ingen indikation af, hvor fejlen er placeret eller hvilken type, der er tale om.

Hvorfor så programmere i maskinkode? Jo, der er skam fordele, der rigeligt opvejer ulemperne, som vi her har ridset op. Den mest nærliggende fordel er den hurtighed, hvormed et maskinkodeprogram udføres. I visse tilfælde er der tale om en forøgelse af hastigheden på flere hundrede gange. Grunden til denne hastighedsstigning er naturligvis, at vi går helt udenom BASIC-fortolkeren og kommunikerer direkte med computerens inderste.

En anden afgørende fordel er, at man kan udrette ting, der i

BASIC forekommer og er umuligt. I maskinkode råder man over hele computerens samlede kapacitet og er ikke kun afhængig af producentens indbyggede programmer (BASIC-fortolker og operativsystem).

Den processor, som vi skal lære at programmere, hedder Z80A, og det er Amstrads CPU (Central Processing Unit). Det er den mest brugte processor indenfor hjemmecomputerindustrien, og den er konstrueret af et firma ved navn Zilog.

Udover denne – for os essentielle – processor indeholder Amstrad flere andre chips, som f.eks. den såkaldte RAM (Random Access Memory). Dette er brugerens hukommelsesområde, og det er her, at alle programmer, variabler, oplysninger om skærmbilledet etc. lagres.

En anden chip har vi allerede omtalt – nemlig ROM (Read Only Memory). Denne er fastlagt af Amstrad's producenter, og man kan ikke ændre dens indhold i modsætning til RAM. ROM indeholder BASIC'en og operativsystemet.

Et ciffer, der indikerer, om der er tale om høj eller lav spænding, kaldes en bit, og den kan altså antage to værdier, nemlig 1 og 0. Man kan sammensætte 8 bit, hvorved der fås en byte. I denne kan lagres et tal mellem 0 og 255. De 8 bit i byten har deres eget nummer og deres egen værdi:

bitnummer	7	6	5	4	3	2	1	0
værdi	128	64	32	16	8	4	2	1

Skal man udtrykke tallet 170, så vil det binært hedde 10101010, nemlig 1 * 128 + 0 * 64 + 1 * 32 + 0 * 16 + 1 * 8 + 0 * 4 + 1 * 2 + 0 * 1. Lader man alle bit i en byte være 1, så fås det højst mulige indhold – den højst mulige værdi – af en byte. Og tallet 11111111 i binær notation vil være 1 * 128 + 1 * 64 + 1 * 32 + 1 * 16 + 1 * 8 + 1 * 4 + 1 * 2 + 1 * 1 = 255. Tilsvarende giver 00000000 i binær notation tallet 0.

Normalt regner vi mennesker i titalsystemet, hvor computeren altså beskæftiger sig med totalsystemet. Imidlertid findes der et tredje talsystem, og det skal vi indføre nu. Det er det hexadecimale talsystem – også kaldet sekstentalsystemet. Et hexadecimalt ciffer kan antage en værdi mellem 0 og 15, og disse cifre hedder

0, 1, 2, 3, ... , 8, 9, A, B, C, D, E og F. A er således det decimale 10 o.s.v.

Vi kan opstille flg. konverteringstabel mellem talsystemerne:

Decimalsystemet Titalsystemet	Binært system Totalsystemet	Hexadecimalsystemet Sekstentalsystemet
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11

Det hexadecimale system bruges i forbindelse med maskinkodeprogrammering, og det skyldes, at man kan udtrykke netop 4 bit i et hexadecimalt ciffer, hvorfor et tal i en byte kan udtrykkes ved hjælp at to hexadecimale cifre. Her vil det første vise antallet af 16'ere og det sidste antallet af 1'ere. AE vil således være 16 * 10 + 14 = 174. Tallet 174 er 10101110 i binær notation. Set 4 bit ad gangen giver det hhv. 1010 og 1110, og det er de to cifre A og E i hexadecimal notation.

Et maskinkodeprogram består således af en række tal alle mellem 0 og 255, og hvert af disse tal – fyldende en byte – kan lagres på én plads i hukommelsen. Da der i RAM er 64 Kbytes (ROM har i øvrigt 32 Kbytes) – det er 65535 bytes – er der altså god plads. Enhver af disse 65535 pladser i hukommelsen kaldes en adresse, og man siger, at man lægger en værdi på en adresse.

Vi skal senere vende tilbage til RAM's opbygning, men her vil

vi blot bringe et i nogen grad uforklaret BASIC-program. Dette program kaldes en hex-loader, og det er med dette redskab, at vi lægger vores programmer bestående af bytes på adresser i RAM. De lægges på en lang række startende fra adresse 40000 i RAM. Da en byte kan udtrykkes ved hjælp af to hexadecimale cifre, skal disse altid indtastes parvis i hex-loaderen. Der må gerne indtastes mange par på en gang, og når der skal afsluttes, skal skrives S. Vi vender i næste kapitel tilbage til brugen af hex-loaderen:

```
10 CLS
20 ADR = 40000 : MEMORY 39999
30 IF H$=""THEN PRINT "INDTAST HEXKODERNE":LINE
   INPUT H$:H$=UPPER$(H$)
40 IF H$="S" THEN END
50 POKE ADR, VAL ("&"+LEFT$(H$,2))
60 H$=RIGHT$(H$,LEN (H$)-2):ADR=ADR+1
70 GOTO 30
```

Vi vil ikke forklare dig programmet, blot bede dig om at gemme det på bånd, da du vil skulle bruge det i alle de kommende maskinprogrammer, der skal indtastes.

Vi afslutter dette kapitel med en lille appetitvækker før den egentlige programmering. Indtast dette BASIC-program:

```
 10 CLS : S = 0 : POKE & 80FF , 2 MEMORY 29999 : PEN 1
 20 FOR N = 40000 TO 40504
 30 READ A $ : S = S + VAL ( " & " + A $ ) : POKE N, VAL ( " &
    "+ A $ )
 40 NEXT
 50 IF S < > 53865 THEN PRINT "FEJL I DATALISTE - RET
    DET!!!":END
52 INPUT "HAR DU EN CPC 664 (eller CPC 6128)? (j/n) ", A$
54 IF INSTR ("jJ", A$) = 0 THEN 58
56 POKE 40159, &D7 : POKE 40160, &B7 : POKE 40163, &D8:
   POKE 40164, &B7 : POKE 40174, &E8 : POKE 40175, &B7 :
   POKE 40155, &36
58 SAVE "DEMO" : CALL 40368
60 REM * * * FARVESTRIBER - 40000 * * *
70 DATA 11,00,00,D5,E1, 3E,02,0E,28,F5,CD,
```

```
DE,BB,06,04,C5,D5,CD,C0,BB,11,00,00,D5,21,8F,01,CD,F9,
BB,E1, D1,13,13, 13,13,C1,10, E8,F1, 3C,FE,10,20,02,
3E,02,0D,20,D7
 80 DATA 3E,00,CD,DE,BB, 11,10,00,21,64,00,E5, D5,D5,E5,CD,
C0,BB,11,6C,02,E1,D5,CD,F6,BB,D1,21,2C,01,E5,CD,
F6,BB,E1,D1,CD,F6, BB,D1,E1,CD,F6,BB
 90 REM *** OVERSKRIFT - 40094 ***
100 DATA 21,0B,06,CD,75,BB,06,2D,21,B1,9C,7E,
CD,5A,BB,23,10,F9,C9, 16,01,4D,41,53,4B,49,4E,4B,4F
,44,45,0A, 0A,08,08,08,08,08, 08,08,4D,45,44,0D,0A,0A,09,
09,09,41,4D,53,54,52,41, 44,20,43,50,43,20, 34,36,34
110 REM *** SCROLLING-FARVER - 40158 ***
120 DATA 11,DC,B1,D5,21,DD,B1,01,0D,00,1A,ED,B0,
12,E1,11,ED,B1,01,0E,00,ED,B0,EB,23,36,FF,C9
130 REM *** SKRÅSKRIFT - 40186 ***
140 DATA 11,20,00,21, 00,94,CD,AB,BB,CD,
06,B9,21,FF,3F,3E,FF,06,E0,C5,01,08,00, 11,FF,93,ED,B8,
E5, EB,23,E5,06,04,CB,2E, 23,10,FB,E1,F5,CD, A8,BB,F1,3D,
E1,C1, 10,E1,C9
150 REM *** SCROLL - 40237 ***
160 DATA 3E,07,01,B0,07,09, E5,01,50,00,ED,B0,
D1,3D,20,F2,C9,11,00, A0,21,00,C0,01,50, 00,C5,D5,E5,
ED,B0,21,50,C0,D1,E5,CD,2D,9D,E1,11,00,
F8,06,18,C5,01,50,00,E5,D5,E5, ED,B0,D1,CD,2D,9D,E1,01,
50, 00,09,EB,E1,09,C1,10, E8,E1,C1,11,80,FF,ED,B0,C9
170 REM *** LEFTSCROLL - 40314 ***
180 DATA 21,90,C1,0E,0F,E5, 3E,50,F5,06,08,C5,E5,D1,23,01,
4F,00, ED,B0,AF,12,01,B0,07,09,C1,10,EE,F1,
3D,E1,E5,20,E5,E1,11, 50,00,19,0D,20,DA,C9
190 REM *** INTERRUPT - 40358 ***
200 DATA 21,FF,80,35,C0,36,02,C3,DE,9C
210 REM *** START - 40368 ***
220 DATA 3E,01,01,18,00,CD,32,BC, 3E,00,CD,0E,BC,CD,FA,
9C,01,00,00,C5, CD,38,BC,C1,AF,CD,32,BC,3E,0E,
0E,18,CD,32,BC,CD,40,9C,11,A6,9D,01,00, 81,21,00,
9F,CD,D7,BC,06,C8, C5,CD,3E,9D,C1,10,F9,CD,7A,9D
230 REM *** SLUTDATA - 40430 ***
240 DATA 21,F7,9D,06,42,CD, A9,9C,C9,1C,01,18,18,1F,01,
08,47,4F,44,20,46,4F,52,4E,30,4A,45,4C,53,45,20,
```

4D,45,44,0A,0A,0D,50,52,4F,47,52,41,4D,4D,45,52,49,4E,47,
45,4E,1F,08,0D,4A,30,52,4E,20,26,20,48,
45,4E,52,49,4B,16,00, 0F,02,1F,01,11

Dette BASIC-program foretager en indkøring af alle de data, der ligger i slutningen af programmet. De bliver lagt på adresser, hvorefter CALL 40368 eksekverer maskinkodeprogrammet, der er indkørt! Der er tale om den rene maskinkode!

Demonstrationsprogrammet udfører ting, som er umuligt i BASIC. Det er udarbejdet som et hurtigt eksempel på, hvad man kan lave i maskinkode. Det er ikke særlig nyttigt, men giver dig forhåbentlig et indblik i maskinkodens muligheder.

Når du har læst denne bog til ende, vil du kunne lave programmer, der udnytter Amstrads fulde potentiel. Nogle af de anvendte rutiner forklares i løbet af bogen, og resten er op til dig selv at forklare ved bogens slutning.

Kapitel 2

Vi skal nu starte på selve programmeringen i maskinkode, og i dette kapitel vil du få den grundlæggende viden om maskinkode-programmering, som er nødvendig. Vi har jo allerede i kapitel 1 introduceret vort vigtigste arbejdsredskab: hex-loaderen. Når vi fremover viser et program, så vil vi bede dig indtaste det i hex-loaderen, og derefter eksekvere det. Vi forventer, at du bruger den viste hex-loader, så der ikke sker fejltagelser undervejs.

Hvis du er vant til at arbejde i BASIC, så ved du, at værdier kan lagres i forskellige variable. Antallet af disse er faktisk ubegrænset. Sådan er det ikke i maskinkode. Her opererer man med registre. Et af disse er det såkaldte A-register (akkumulatoren). Dette er et 8 bits-register, hvilket, som du med din nuværende viden formentlig vil kunne slutte dig til, betyder, at det kan indeholde et tal mellem 0 og 255. 255 er jo netop 11111111 i binær notation. Z80-processoren har ikke så mange registre, at man som programmør kan tillade sig at rutte med pladsen. Umiddelbart findes der kun 7 brugbare 8 bits-registre, d.v.s. 7 »pladser« til at gemme et tal mellem 0 og 255. Disse registre hedder foruden A B, C, D, E, H og L. Alle har altså hver deres genkendelige bogstav. Disse er indført for, at vi mennesker bedre kan programmere i maskinkode.

Hvad nu, hvis man vil bruge tal over 255? Heldigvis er processoren indrettet til dette også. Man kan nemlig tage B- og C-registrene og sætte sammen til et 16 bits-register. I dette kan man så gemme tal helt op til 65535, og som vi allerede har nævnt er det det største tal, som Z80 kan arbejde med. Ialt kan der sammensættes 3 sådanne 16 bits-registre. Disse kaldes BC, DE og HL. Ligesom BC-registret er sammensat af B og C, vil DE være sammensat af D- og E-, og HL af H- og L-registrene.

I 16 bits-registrene kan man altså registrere værdier op til 65535. Ønsker vi f.eks. at gemme tallet 10000 i DE-registret, så gøres det ved, at D-registret indeholder antallet af 256'ere i tallet,

E-registret antallet af 1'ere. I dette tilfælde vil D indeholde 39, E vil indeholde 16, og sammensat til et 16 bits-register vil DE-registret derfor indeholde 256 * 39 + 16 = 10000.

Indtil videre har vi altså 7 8 bits-registre, hvoraf de 6 kan sammensættes til 3 16 bits-registre. A-registret kan ikke sammensættes med noget register og er derfor det, som vi kalder et decideret 8 bits-register.

Når man skal programmere i maskinkode, så opererer man udelukkende med tal – alle mellem 0 og 255. Imidlertid vil det være ret forvirrende at se et maskinkodeprogram udelukkende bestående af en hel flok tal uden et eneste genkendeligt tegn. Derfor har man indført det, der kaldes mnemonics. Disse er navne på forskellige maskinkodeinstruktioner, og på den måde kan vi lettere overskue et program. Mnemonic'en, huskeordet, hvormed vi kan give et register en værdi, hedder LOAD (forkortet LD). Hvis vi ønsker at give A-registret værdien 174 (10101110 i binær notation), så hedder mnemonic'en

```
LD A , 174
```

Den instruktion, der giver A-registret en vilkårlig værdi, hedder derfor generelt

```
LD A , n
```

hvor n er et tal mellem 0 og 255. Denne mnemonic kan man jo ikke blot indtaste i vores hex-loader, så vi skal have konverteret den til et tal, der angiver, at det er netop instruktionen LD A , n, som vi ønsker at arbejde med (en assembler ville ikke kræve denne konvertering, da man med denne arbejder med mnemonics). Hvis du slår op i vort appendix A bag i bogen, så vil du under hex-kode 3E finde netop denne mnemonic. 3E (62 i decimal notation) er altså det tal, der angiver, at vi nu vil tildele A-registret en værdi mellem 0 og 255, men hvilken værdi? Den værdi, som registret skal tildeles, skal stå i byten bagefter. Det betyder, at denne instruktion vil fylde to bytes, nemlig en byte, der angiver, at der er tale om instruktionen, der tildeler A en værdi, og en byte, der angiver denne værdi. Ønsker vi at tildele A-registret værdien 174

(det er AE i hexadecimal notation), så vil hex-koden, som koden for den oversatte mnemonic hedder, hedde 3EAE.

Selv om du nu kun er blevet præsenteret for en enkelt instruktion af alle de, der er i Z80-instruktionssættet, så skal vi allerede nu afprøve denne.

Når vi laver et program i maskinkode, kaldes det en rutine. Når man ønsker at køre dette program, sker det ved, at man kalder den adresse, hvor programmet starter. Hvis vi laver et maskinkodeprogram og placerer det fra adresse 40000, så vil vi kunne køre programmet med

```
CALL 40000
```

som BASIC-kommando.

På samme måde kan man i maskinkodeprogrammer kalde rutiner, som producenten har lavet (disse rutiner ligger i ROM). Vi vil i det følgende program udnytte en rutine, fastlagt af producenten, der udskriver en karakter givet ved værdien i A-registret. Hvis A-registret indeholder værdien 65, så vil denne rutine skrive et "A" på skærmen, da bogstavet A har karakterkoden 65 (se evt. din instruktionsbog). Indholder A-registret værdien 66, så udskrives B o.s.v.

Disse rutiner kan altså kaldes fra maskinkode, og det sker med CALL-instruktionen. For at få en uddybende forklaring af denne instruktion må du vente til kapitel 9. Desuden vil vi her postulere, at man ved at bruge RET-instruktionen kan komme tilbage fra det kørende maskinkodeprogram til BASIC. Dette vil også blive uddybet i kapitel 9.

Nu vil vi lave vores første lille rutine baseret på instruktionen LD A,n.

Hex-kode	**Mnemonic**
3E41	LD A , 65
CD5ABB	CALL TXT OUTPUT
C9	RET

Hele denne forenklede rutine fylder 6 bytes. Først giver vi register A værdien 65 (hexkoden hedder 3E41 som før beskrevet), så

kalder vi den rutine i ROM, der udskriver karakteren med koden i A-registret, og endelig returnerer vi fra vores lille maskinkode-program til BASIC.

Du finder nu din hex-loader frem og indtaster i henhold til vores forklaringer i kapitel 1:

» 3E41CD5ABBC9S «

Hvis du betragter hex-loaderen, så vil du kunne se, at vi har skrevet

MEMORY 39999

Hvilket betyder, at pladserne fra 40000 og knapt et par tusinde bytes frem er forbeholdt os til vore rutiner. Vi har nu ved hjælp af hex-loaderen lagt vort program ind på adresserne 40000, 40001, 40002, 40003, 40004 og 40005. Vi kalder vores rutine (d.v.s. kører programmet) med

CALL 40000

– hvilket forhåbentlig resulterede i udskriften af et A (CHR $ (65)). Kontrollér selv i instruktionsbogen.

På samme måde kan man tildele de andre 8 bits-registre værdi. Her er listen over alle disse instruktioner:

Hex-kode	**Mnemonic**
06n	LD B,n
0En	LD C,n
16n	LD D,n
1En	LD E,n
26n	LD H,n
2En	LD L,n
3En	LD A,n

hvor n igen angiver, at der er tale om et 8 bits-tal.

Desuden kan vi give et 16 bits-register en værdi. Instruktionen

LD BC,10000

giver naturligvis BC-registret værdien 10000, og som tidligere beskrevet vil B-registret indeholde antallet af 256'ere og C-registret antallet af 1'ere. B indeholder derved 39, og C indeholder resten af de 10000, nemlig 16.

Alle tre 16 bits-registre kan tildeles en værdi (LOAD'es) på denne måde.

Hex-kode	**Mnemonic**
01nn	LD BC,nn
11nn	LD DE,nn
21nn	LD HL,nn

hvor nn angiver, at der er tale om et 16 bits-tal.

Disse instruktioner fylder altså 3 bytes, nemlig en byte, der angiver selve instruktionens betydning, og de to bytes, der viser hvilket tal, der skal LOAD'es ind i registret.

Antallet af 256'ere (der kommer i B-registret, hvis vi LOAD'er BC-registret) skal være det sidste tal i hex-koden, mens antallet af 1'ere vil være det første af de to bytes, der indicerer tallet.

Skal BC LOAD'es med 10000, vil hex-koden være 011027, idet der er 39 (27 i hexadecimal notation) 256'ere og 16 (10 i hexadecimal notation) 1'ere.

Desuden findes der LOAD-instruktioner, der kopierer en værdi fra et register i et andet register. For eksempel vil

```
LD A,B
```

kopiere værdien i B i A-registret, så de nu indeholder den samme værdi, nemlig B-værdien. A-registrets gamle værdi forsvinder i tågerne.

Dette kan gøres imellem alle 8 bits-registrene, som vi har beskrevet. Værdien i A kan således blive kopieret i både B, C, D, E, H, L og sågar også i A-registret selv, hvad nytte man så end har af det?

Alle hex-koderne for disse kopiinstruktioner kan iagttages i appendiks A, så vi har fundet det meningsløst at skrive dem alle op her.

Med disse instruktioner kan vi ændre vort program fra før, og dermed verificere vores nuværende viden.

Hex-kode	**Mnemonic**
0643	LD B,67
78	LD A,B
CD5ABB	CALL TXT OUTPUT
C9	RET

Indtast det i hex-loaderen og kald det med CALL 40000.

Hvad blev resultatet? Et C? Forhåbentlig. Kan du se hvorfor? Vi LOAD'er B-registret med 67 (karakterkoden for bogstavet C), og dette kopieres så i A-registret, hvor det ligger, når vi kalder rutinen, der udskriver karakteren.

Prøv selv med andre rutiner. Find for eksempel karakterkoden for et bogstav, læg den i et register (prøv E-registret), og kopiér det så i A-registret og kald ROM-rutinen TXT OUTPUT. Husk også RET. Så har du faktisk lavet dit første maskinkodeprogram . . .

Vi mangler lige at nævne, at der desværre ikke findes instruktioner i Z80-sproget til kopiering af 16 bits-registre. Hvis DE-registrets indhold skal kopieres i BC, må man bruge

```
LD B,D
LD C,E
```

da en instruktion som

```
LD BC,DE
```

ikke findes.

Kapitel 3

Man kan også i maskinkode finde indholdet på en bestemt adresse. Hvis man f.eks. ønsker indholdet på adresse 40000 lagt ind i A-registret, så bruges instruktionen

```
LD A, (40000)
```

Bemærk, at vi her benytter parenteser rundt om adressen. Det betyder, at vi nu ikke arbejder med en talværdi, men derimod med indholdet på adressen, som talværdien angiver. Vi kan afprøve det i et eksempel:

Hex-kode	**Mnemonic**
3A409C	LD A , (40000)
CD5ABB	CALL TXT OUTPUT
C9	RET

CALL 40000 efter indtastning af rutinen i hex-loaderen vil give udskriften af CHR $ (58). Kan du se hvorfor? På adresse 40000 står første byte i vort program, og det er 3A, der er 58 i decimal notation. Derfor vil rutinen, vi kalder, udskrive karakteren med koden 58.

A-registret er det eneste af 8 bits-registrene, der direkte kan finde indholdet af en givet adresse. Bliver adressen i stedet for at blive indtastet sammen med 3A lagt i HL-registret, så kan alle 8 bits-registre klare opgaven. Vi laver derfor et nyt eksempel:

Hex-kode	**Mnemonic**
21409C	LD HL,40000
56	LD D,(HL)
7A	LD A,D
CD5ABB	CALL TXT OUTPUT
C9	RET

CALL 40000. Programmet er selvfølgelig langtfra så enkelt som muligt, men det er ligegyldigt. Formålet i dette og mange af de kommende programmer er blot at forstå de brugte instruktioner. I denne rutine LOAD'es HL først med tallet 40000. Derefter LOAD'es 8 bits-registret D med indholdet på adresse 40000 (HL indeholder jo netop 40000). Denne værdi overføres så til A, karakteren med den kode udskrives, og der returneres til BASIC. Viste skærmen CHR $(33)?

De to andre 16 bits-registre, som vi har beskrevet, kan kun overføre indholdet på en adresse til A-registret. Det giver alt i alt disse instruktioner:

Hex-kode	**Mnemonic**
3Ann	LD A,(nn)
0A	LD A,(BC)
1A	LD A,(DE)
7E	LD A,(HL)
46	LD B,(HL)
4E	LD C,(HL)
56	LD D,(HL)
5E	LD E,(HL)
66	LD H,(HL)
6E	LD L,(HL)

Du kan selv prøve at eksperimentere videre med dem, og derved blive fortrolig med disse instruktioner. De er essentielle for den videre programmering.

Vi har tidligere beskæftiget os med 3 16 bits-registre, men Z80-processoren indeholder flere endnu. To af dem er de såkaldte indeks-registre, nemlig IX- og IY-registrene. Disse er meget specielle, hvilket vi nu vil gøre rede for.

Først skal vi nævne, at vi kan give dem en værdi med

Hex-kode	**Mnemonic**
DD21nn	LD IX,nn
FD21nn	LD IY,nn

Bemærk, at disse hex-koder fylder ialt 4 bytes. Det er faktisk koden for instruktionen LD HL,nn, der er blevet påsat hhv. DD og

FD. Sådan vil det imidlertid altid være for de to indeks-registre. Alle instruktioner med IX-registret består af DD + den tilsvarende kode for HL-registret, og for IY-registret består den af FD + den tilsvarende instruktion for HL-registret. De er som før nævnt alle opført i appendiks A.

Når man har givet IX en værdi (IY ligeledes, naturligvis), så kan man bruge denne værdi som udgangspunkt for sine operationer og derefter regere i et område af hhv. 128 bytes tilbage og 127 bytes frem. Vi tager et eksempel:

Hex-kode	**Mnemonic**
DD21409C	LD IX,40000
DD7E01	LD A,(IX + 1)
CD5ABB	CALL TXT OUTPUT
C9	RET

CALL 40000. Resultatet blev igen CHR $ (33).

Først giver vi index-registret IX værdien 40000, og dernæst udfører vi instruktionen, som svarer til

LD A,(40001)

idet IX jo er 40000. Herved får A-registret værdien, der står på adresse 40001, der er den anden byte i vor rutine. Her står netop 33. Den udskrives så som en karakter, og der returneres.

Denne form for registre er meget nyttige, når man skal operere i en forholdsvis beskeden radius rundt om en given adresse. Instruktionerne findes i stort omfang, både for IX- og IY-registrene. Der findes således disse, når man ønsker at finde indholdet på en adresse:

Hex-kode	**Mnemonic**
DD7Ed	LD A,(IX + d)
FD7Ed	LD A,(IY + d)
DD46d	LD B,(IX + d)
FD46d	LD B,(IY + d)
DD4Ed	LD C,(IX + d)
FD4Ed	LD C,(IY + d)
DD56d	LD D,(IX + d)

FD56d	LD D,(IY + d)
DD5Ed	LD E,(IX + d)
FD5Ed	LD E,(IY + d)
DD66d	LD H,(IX + d)
FD66d	LD H,(IY + d)
DD6Ed	LD L,(IX + d)
FD6Ed	LD L,(IY + d)

Vi kan altså også lade d antage en negativ værdi, og det gøres med den såkaldte to-komplement-metode. En tabel for denne ser således ud:

Tokompl.	Binært	Decimalt
− 128	10000000	128
− 127	10000001	129
− 126	10000010	130
. . .		
− 2	11111110	254
− 1	11111111	255
0	00000000	0
1	00000001	1
2	00000010	2
. . .		
126	01111110	126
127	01111111	127

Tallet 10000000 i binær notation vil normalt angive værdien 128, men to-komplementært angiver det − 128, hvilket ses i tabellen. Til venstre i denne ses således to-komplementet, mens det normale tal står til højre. Der ses, at man sagtens kan udtrykke negative tal, idet man lægger en begrænsning og indfører de førnævnte − 128 og 127 som yderværdier. Negative tal udtrykkes da som 256 minus det tal, man ønsker at gå tilbage. Hex-koden for LD A,(IX − 5) vil da være DD7EFB, idet FB er 251 (256 − 5) i decimal notation.

Vi tager et eksempel:

Hex-kode	**Mnemonic**
DD21A49C	LD IX,40100

DD7E9D	LD A,(IX – 99)
CD5ABB	CALL TXT OUTPUT
C9	RET

Med CALL 40000 fås igen resultatet CHR $ (33). Her bliver IX LOAD'et med 40100, og A LOAD'es så med indholdet på adresse 40100 – 99 = 40001, og det er atter 33. Bemærk, at hex-koden for LD A,(IX – 99) blev DD7E9D, hvor 9D er lig tallet 157, nemlig 256 – 99. To-komplementært udtrykker det dog stadig – 99.

Som du ser, er det lidt mere besværligt end normalt at arbejde to-komplementært, og umiddelbart behøver du ikke arbejde i dette system. Det bliver dog nødvendigt, når du benytter indeks-registrene som vist, og i kapitel 6 skal vi vise nye eksempler på brugen af to-komplement-metoden, hvor det vil være svært at undvære det fornødne kendskab til dette.

Man kan også finde værdien af et tal mellem 0 og 65535 gemt på to adresser.

LD HL,(nn)

vil således finde indholdet på adresse nn + 256 gange indholdet på adresse (nn + 1) i HL-registret. Det samme vil kunne laves med de andre 16 bits-registre. Ialt fås:

Hex-kode	**Mnemonic**
ED4Bnn	LD BC,(nn)
ED5Bnn	LD DE,(nn)
2Ann	LD HL,(nn)
DD2Ann	LD IX,(nn)
FD2Ann	LD IY,(nn)

Som det ses, er det ikke blot instruktioner hidrørende fra IX- og IY-registrene, der er på 4 bytes. BC- og DE-instruktionerne er også 4 bytes lange. Det skyldes, at ikke alle instruktioner vil kunne bruges umiddelbart (der er jo kun plads til 256 umiddelbart), og derfor åbner ED – og CB som vi senere skal se – for andre instruktioner. I appendiks A står disse opført under »Efter EDh : «.

Vi laver nu et eksempel til illustration af ovenstående. Når vi i programmet skriver forkortelsen DEFB, betyder det blot, at hex-

koderne ikke skal forstås som maskinkodeinstruktioner, men derimod bare er nogle bytes, som programmet bruger som data. DEFB står for DEFine Byte.

Hex-kode	**Mnemonic**
2A489C	LD HL,(40008)
7E	LD A,(HL)
CD5ABB	CALL TXT OUTPUT
C9	RET
409C	DEFB – 40000 –

CALL 40000. Blev resultatet CHR $ (42)? Vi starter med at LOAD'e HL med indholdet på 40008 + 256 gange indholdet på 40009 (H får indholdet på 40009, L det på 40008), og dette er 40000, idet disse adresser netop er de to, vi i programmet kalder DEFB. A-registret LOAD'es nu med indholdet på denne adresse (40000), og her står programmets første byte, og det er 42. Karakteren med koden 42 udskrives, og der returneres til BASIC.

Mulighederne med alle disse instruktioner er store, men i praksis vil man ofte benytte HL-registret, da det som allerede vist er det register af alle 16 bits-registrene, der levner brugeren det største albuerum.

Det modsatte af at aflæse indholdet på en adresse er at lægge en værdi på en adresse. Hvis man ønsker at lægge indholdet i A-registret på en adresse, så gøres det med

LD (nn),A

Ligesom før viser parentesen, at der er tale om en adresse, og nn indicerer naturligvis, at der er tale om et 16 bits-tal. Vi afprøver det i et eksempel:

Hex-kode	**Mnemonic**
3E41	LD A,65
32469C	LD (40006),A
C9	RET

CALL 40000, og skriv så derefter PRINT PEEK (40006). PEEK-funktionen i BASIC giver jo indholdet på den angivne adresse. Nu skulle svaret gerne være 65.

Andre måder at placere værdier på er ved at lægge den adresse, hvorpå værdien skal lagres, i ét register, og værdien, der skal placeres dér, i et andet. Mnemonic'en LD (HL),A betyder derfor, at værdien i A-registret lægges på den adresse, HL indeholder.

Eksempel:

Hex-kode	**Mnemonic**
21479C	LD HL,40007
3E43	LD A,67
77	LD (HL),A
C9	RET

CALL 40000 : PRINT PEEK (40007) skulle gerne give resultatet 67.

Som da vi hentede værdier på adresser, er HL også her det mest begunstigede 16 bits-register. A-registret er, som det ses herunder, det mest begunstigede 8 bits-register. Ialt findes der disse instruktioner, der overfører én værdi til én adresse:

Hex-kode	**Mnemonic**
32nn	LD (nn),A
77	LD (HL),A
70	LD (HL),B
71	LD (HL),C
72	LD (HL),D
73	LD (HL),E
74	LD (HL),H
75	LD (HL),L
02	LD (BC),A
12	LD (DE),A
36n	LD (HL),n

Vi tager endnu et eksempel, der viser brugen af nogle af de instruktioner, som vi har lært indtil nu.

Hex-kode	Mnemonic
2A4A9C	LD HL,(40010)
4E	LD C,(HL)
79	LD A,C
114A9C	LD DE,40010
12	LD (DE),A
C9	RET
409C	DEFB – 40000 –

CALL 40000 : PRINT PEEK (40010). Gå selv rutinen igennem, og find ud af, hvorfor resultatet blev 42.

Vi kan også lægge en værdi ind på en adresse givet en vis afstand fra enten IX- eller IY-registrets værdi. Der findes ialt:

Hex-kode	Mnemonic
DD77d	LD (IX + d),A
DD70d	LD (IX + d),B
DD71d	LD (IX + d),C
DD72d	LD (IX + d),D
DD73d	LD (IX + d),E
DD74d	LD (IX + d),H
DD75d	LD (IX + d),L
DD36dn	LD (IX + d),n
FD77d	LD (IY + d),A
FD70d	LD (IY + d),B
FD71d	LD (IY + d),C
FD72d	LD (IY + d),D
FD73d	LD (IY + d),E
FD74d	LD (IY + d),H
FD75d	LD (IY + d),L
FD36dn	LD (IY + d),n

Vi vil ikke bringe nogle eksempler på disse instruktioner. Alligevel beder vi dig notere en helt speciel, men konsekvent notation, som indeks-registrene benytter. I en instruktion som LD (IX + d),n skal der bruges 4 bytes. En, der angiver, at man bruger IX-registret, en der angiver instruktionens art, en der fortæller hvor stor afstand fra IX vi bevæger os i (d), og endelig værdien (n), som skal lagres. I sådanne instruktioner vil distancen fra IX-registrets vær-

di, kaldet d, *altid* være 3. byte i hex-koden. Således også her, hvor værdien n er 4. byte.

Naturligvis kan det også lade sig gøre at lægge 16 bits-tal ind på to adresser. Skal værdien i HL-registret lagres på en adresse nn, så bliver L lagret på adresse nn og H på adresse (nn + 1), idet H, der jo indeholder antallet af 256'ere, lagres sidst – i fuld overensstemmelse med vore tidligere anvisninger. Instruktionen for denne transaktion hedder LD (nn),HL.

Vi prøver den i et eksempel:

Hex-kode	**Mnemonic**
211027	LD HL,10000
22479C	LD (40007),HL
C9	RET

CALL 40000 : PRINT PEEK (40007) + 256 * PEEK (40008) giver forhåbentlig svaret 10000. Vi har altså lagret antallet af 1'ere (kaldet LSB af tallet – LSB (Least Significant Byte) betyder den mindst vigtige byte) på adresse 40007, og antallet af 256'ere (kaldet MSB (Most Significant Byte) for den mest vigtige byte) på adresse 40008.

Man kan også bruge andre registre. Der findes:

Hex-kode	**Mnemonic**
ED43nn	LD (nn),BC
ED53nn	LD (nn),DE
22nn	LD (nn),HL
DD22nn	LD (nn),IX
FD22nn	LD (nn),IY

Vi bringer her en fuldstændig oversigt over alle registrene i Z80-processoren.

A	F	A'	F'
B	C	B'	C'
D	E	D'	E'
H	L	H'	L'

IX	
IY	
SP	
PC	
I	R

De otte registre til højre i skemaet kaldes de alternative registre, og dem vender vi først tilbage til i kapitel 19. SP er den såkaldte Stack Pointer, og den vender vi tilbage til i kapitel 8. PC er Program Counter, og den beskæftiger vi os med i kapitel 6.

Tilbage står tre registre vi endnu ikke har omtalt, nemlig F, I og R-registret. I-registret er det såkaldte Interrupt-register, der bruges til behandling af et interrupt. Vi vil i kapitel 18 se lidt nærmere på interrupt i al almindelighed og således her nævne I-registret. R-registret er det såkaldte Refresh-register, der bruges til genopfriskning af RAM. Dette register vil brugeren næppe kunne få brug for.

Der findes kun 4 instruktioner til disse to registre:

Hex-kode	**Mnemonic**
ED57	LD A,I
ED5F	LD A,R
ED47	LD I,A
ED4F	LD R,A

F-registret er ét af de mest interessante i Z80-processoren. Det er det såkaldte flag-register, og det skal ikke opfattes som et register, man kan adressere. Dette er nemlig ikke muligt umiddelbart, og man har ikke gavn af det på den måde. Flag-registret er derimod maskinkodeprogrammørens vigtigste værktøj til at afgøre for-

skellige tvivlssager; om et tal er negativt, om et tal er for stort til at være i én byte, om et register er nul m.v. Hver bit i flag-registret har sin betydning. Hver bit ud af de 8 (F-registret er et 8 bits-register) har sit eget navn og kaldes for et flag. Det kan indtage to tilstande, nemlig 1 eller 0. Er bitten 1, siger man, at flaget er sat, og er den 0, er det resat.

Vi vil her bringe en oversigt over flag-registrets bit og fortælle, hvornår de hhv. sættes og resættes. Dette vil først blive aktuelt lidt senere under programmeringen, men vi har fundet det vigtigt, at du allerede nu lærer flagene at kende og stifter bekendtskab med deres særheder.

Opbygningen ser således ud:

Bitnummer	7	6	5	4	3	2	1	0
Flag	S	Z		H		P/V	N	C

Bit 7 er altså det såkaldte S-flag. Det betyder Sign-flag. Denne bit udtrykker, hvorvidt et tal er blevet negativt. Vi har allerede set det under to-komplement-metoden. Hvis du blader tilbage kan du se, at et tal er negativt, hvis bit 7 (bitten længst til venstre) er 1. Det passer med, at man to-komplementært kun kan vise positive tal op til 127 (det er 01111111 i binær notation). Bliver tallet større, sættes bit 7, og tallet angives derved at være negativt. S-flaget kopierer denne bit 7, når en instruktion er udført. Som de andre brugbare flag vil S-flaget ikke altid blive påvirket af en operation. Det vil med andre ord kun tage bit 7's værdi, hvor Zilog (producenten af Z80-processoren) har fundet det passende, og det er da heldigvis de samme steder, som vi formentlig ville finde det passende, hvis det var op til os at vælge. Det sker primært efter addition og subtraktion med registre, hvilket vi vender tilbage til i kapitel 5. Her vil resultatet af operationen lade sin bit 7 – kaldet fortegnsbitten – kopiere i S-flaget. Dette sker også ved de logiske operationer, som vi ser på i næste kapitel.

Bit 6 i flag-registret er det såkaldte Z-flag, hvilket står for Zero-flag. Dette flag sættes, når en operation medfører resultatet nul. Det er f.eks. anvendeligt, når man skal udføre en operation et antal gange og derfor lader en tæller gå mod nul. Herved kan Z-flaget testes, og løkken kan fortsætte, hvis Z-flaget er resat.

Bit 5 – og også bit 3 – i F-registret er uden navn, og de har kun

intern betydning for Z80-processoren. Deres status kan ikke testes af os brugere, og vi kan derfor ikke benytte dem.

Bit 4 er H-flaget, stående for Halfcarry-flaget. Dette flag er heller ikke testbart, hvilket vi vender tilbage til. Det er derfor ikke særlig nyttigt. Blot kan vi fortælle, at det sættes, når hhv. en addition og subtraktion medfører et lån fra bit 4. Flaget bruges til hjælp ved DAA-instruktionen, som forklares i kapitel 11. Brugeren har dog ikke brug for dette flag – vi nævner det blot for fuldstændighedens skyld.

Bit 2 er det såkaldte Parity/oVerflow-flag kaldet P/V-flaget. Dette har, som navnet antyder, to formål. Ved logiske operationer, hvor de enkelte bit behandles, vil det virke som paritetsindikator og derfor vil det hhv. blive sat, hvis resultatet indeholder et lige antal satte bit, og resat for et ulige antal. Ved addition og subtraktion virker det som overflowindikator, der sættes, hvis resultatet medfører overflow. Dette fremkommer, når resultatet af en addition (eller subtraktion) er forkert to-komplementært. Vi vender tilbage til dette fænomen under vor gennemgang af disse instruktioner.

Bit 1 er N-flaget, og det er ikke brugbart. Imidlertid er dets funktion ganske klar, idet det sættes, når den sidst udførte operation er en subtraktion.

Det sidste flag, fra bit 0, er C-flaget stående for Carry-flaget. Sammen med Z-flaget er det det mest benyttede til test af resultater. Flaget sættes, når en addition (eller subtraktion) medfører en mente. Hvis resultatet af en addition således overstiger 255, vil C-flaget sættes for at vise, at der egentlig mangler en 9. bit.

Vi skal senere vise eksempler på, at fire af flagenes tilstande kan bestemme, hvilke operationer der skal udføres. Det er først og fremmest Z- og C-flaget, men også S- og P/V-flaget. F-registret er derfor udslaggivende for mange vigtige afgørelser, og dets indhold vil altså afhænge af, hvad den sidste udførte instruktion var, og hvordan resultatet tog sig ud for de enkelte bit.

LD-instruktionerne, som vi har gennemgået i dette og det forrige kapitel, påvirker ikke flagene overhovedet. Der er dog undtagelser, nemlig

LD A,I

og

LD A,R

hvor S- og Z-flaget sættes/resættes afhængigt af resultatet. Z-flaget sættes således, hvis der i I-registret (eller R-registret) står tallet nul, mens S-flaget sættes, hvis det indhentede tal er negativt tokomplementært.

Når vi i øvrigt nævner fremover, at en instruktion berører eller ikke berører flagene, så menes altid kun de fire førnævnte anvendelige flag: S, Z, P/V og C.

Kapitel 4

I de to foregående kapitler lærte du at ændre værdien i et register med LD-instruktionen. Nu skal du lære at ændre bittene i ét af 8 bits-registrene A, B, C, D, E, H eller L eller på en adresse givet ved (HL), (IX + d) eller (IY + d).

Med instruktionen SET sættes en bit i et af ovenstående registre (eller på en adresse). Hvis register B indeholder 01010101 i binær notation (det er 85 i decimal notation) og instruktionen SET 5,B bliver udført, vil B indeholde 01110101 = 117, idet bit 5 er den sjette bit fra højre. Bit 7 er bitten længst til venstre, bit 0 den længst til højre. Hvis HL indeholder 30000, og SET 0,(HL) udføres, sættes bit 0 i værdien liggende på adresse 30000. Bit 0 i HL-registret sættes således ikke!

På tilsvarende måde kan man resætte en bit med instruktionen RES (RESet). Hvis D-registret indeholder 00010000 binært (16 decimalt), og RES 4,D udføres, indeholder D-registret derefter værdien nul.

Vi tager et konkret programeksempel:

Hex-kode	**Mnemonic**
3E53	LD A,83
CBDF	SET 3,A
CD5ABB	CALL TXT OUTPUT
CBA7	RES 4,A
CD5ABB	CALL TXT OUTPUT
C9	RET

CALL 40000.

A indeholder binært 01010011 (83), og når bit 3 med instruktionen SET 3,A sættes, ændrer A-registret udseende til 01011011 (91). Derefter kaldes rutinen TXT OUTPUT, og karakteren med denne kode udskrives. Derefter resættes bit 4, så A indeholder 01001011 (75 decimalt). Denne karakter udskrives, hvorefter der

returneres. Kontrollér selv ved hjælp af din instruktionsbog, at udskriften er korrekt.

SET- og RES-instruktionerne påvirker ikke flagene. Ialt findes der 80 instruktioner af hver, og du kan se dem alle i appendiks A. Bemærk i øvrigt, at alle hex-koder til disse instruktioner starter med CBh.

Udover at sætte og resætte bits kan man teste, om en givet bit er sat. Dette gøres med instruktionen BIT, der selvfølgelig virker på de samme registre og adresser som SET og RES. BIT-instruktionen virker på den måde, at Z-flaget bliver sat, hvis den testede bit var 0 (bitten var resat), mens Z-flaget resættes, hvis den testede bit var 1 (sat). Zero-flaget er således stedet, hvor det kan ses, om bitten var 0 eller 1, idet det altid indtager den modsatte værdi af den testede bit.

Hvis E-registret er 10011101 og BIT 4,E udføres, vil Z-flaget blive resat, da bit 4 i E-registret (den 5. bit fra højre) er sat.

Der findes også 80 instruktioner af denne slags, og også disse skal indledes med CBh i hex-koden. Vi vil ikke give noget eksempel på brugen af BIT-instruktionen, da vi endnu ikke har vist, hvordan Z-flagets udseende kan behandles.

Nu skifter vi emne, idet du nu skal lære om Z80's logiske instruktioner. Der findes 3 af slagsen, og de hedder AND, OR og XOR (eXclusive OR). Disse opererer med de enkelte bit i register A og et andet register (A, B, C, D, E, H og L), indholdet på en adresse ((HL), (IX + d) og (IY + d)), eller blot et tal n.

Det fungerer på den måde, at når en logisk operation udføres, vil de enkelte bit blive sammenlignet. Hvis instruktionen AND B udføres, vil bit 0 i A blive sammenlignet med bit 0 i B, bit 1 i A med bit 1 i B o.s.v. op til bit 7 i A og bit 7 i B. Når to bit sammenlignes, vil resultatet blive 0 eller 1 afhængig af bittenes værdi. Resultatet af sammenlægningen lægges automatisk i A-registret.

Benyttes AND-instruktionen ser resultatet ud efter nedenstående sandhedstabel:

0 og 0 sammenlignes – resultat: 0
1 og 0 sammenlignes – resultat: 0
0 og 1 sammenlignes – resultat: 0
1 og 1 sammenlignes – resultat: 1

Resultatet er altså kun 1, hvis begge bits er 1. Vi tager et eksempel:

Register A	0	0	1	1	0	1	0	1	= 53
163	1	0	1	0	0	0	1	1	= 163
AND 163	0	0	1	0	0	0	0	1	= 33

A-registret indeholdende 53 sammenlignes med 163, og som det fremgår, vil register A efter sammenligningen (efter instruktionen AND 163) indeholde 33. Kun bit 0 og bit 5 er blevet 1, da både bit 0 i A og bit 0 i 163 er sat samtidig med, at bit 5 i A og 163 er sat.

Vi kan desuden indføre en tommelfingerregel, der siger, at resultatet ved en AND-instruktion ikke bliver større end den mindste af de to sammenlignede værdier.

Vi kan se AND-instruktionen i et eksempel:

Hex-kode	**Mnemonic**
3E35	LD A,53
E6A3	AND 163
CD5ABB	CALL TXT OUTPUT
C9	RET

CALL 40000. Som vi allerede har gjort rede for, bliver resultatet af sammenligningen 33, og karakteren med den kode udskrives, når TXT OUTPUT kaldes.

Der findes følgende muligheder med AND:

Hex-kode	**Mnemonic**
A7	AND A
A0	AND B
A1	AND C
A2	AND D
A3	AND E
A4	AND H
A5	AND L
A6	AND (HL)
DDA6	AND (IX + d)

FDA6	AND (IY + d)
E6n	AND n

AND bruges således mest til at resætte de enkelte bit. Ønsker man kun at bruge de 4 mindst betydende bit (bit 3, 2, 1 og 0) i A-registret, så udføres blot AND 00001111 (AND 15), hvorefter disse 4 beholder deres værdi, mens de 4 mest betydende bliver resat.

Efter AND-instruktionen vil C-flaget være resat. Derfor bruger man tit AND A til at slette carry-flaget med, da denne instruktion ikke ændrer indholdet i register A. Z-flaget vil blive sat, hvis resultatet af sammenligningen bliver nul. S-flaget kopierer bit 7 i A, og P/V-flaget virker som paritetsindikator. Det sættes derfor, hvis sammenligningen medfører et lige antal satte bit.

Den anden logiske instruktion hedder OR, og den minder om AND-instruktionen. Dens sandhedstabel ser blot lidt anderledes ud:

0 og 0 sammenlignes – resultat: 0
1 og 0 sammenlignes – resultat: 1
0 og 1 sammenlignes – resultat: 1
1 og 1 sammenlignes – resultat: 1

Resultatet af sammenligningen for hver enkelt bit bliver således 1, hvis blot én af de to bit er 1. Eksempel:

Register A	0	1	1	0	0	1	1	1	= 103
Register D	1	0	1	0	0	1	0	0	= 164
OR D	1	1	1	0	0	1	1	1	= 231

Register A indeholdende 103 sammenlignes med register D, der indeholder 164, og det samlede resultat er 231, som det fremgår herover.

Vores tommelfingerregel ved OR er, at resultatet aldrig kan blive mindre end den største værdi af de to sammenlignede. Sammenligningen herover kan ses i dette programeksempel:

Hex-kode	Mnemonic
3E67	LD A,103
16A4	LD D,164
B2	OR D
CD5ABB	CALL TXT OUTPUT
C9	RET

Resultatet af denne rutine bliver derfor udskrift af CHR $ (231).

Der findes flg. OR-instruktioner:

Hex-kode	Mnemonic
B7	OR A
B0	OR B
B1	OR C
B2	OR D
B3	OR E
B4	OR H
B5	OR L
B6	OR (HL)
DDB6	OR (IX + d)
FDB6	OR (IY + d)
F6n	OR n

Man kan således med OR-instruktionen sætte flere bit. OR 11110000 (OR 240) vil således sætte de fire mest betydende bit, mens de 4 mindst betydende vil beholde deres værdi. Flagene påvirkes på samme måde ved OR- som ved AND-instruktionen.

Den tredje logiske instruktion hedder XOR, og den har følgende sandhedstabel:

0 og 0 sammenlignes – resultat: 0
1 og 0 sammenlignes – resultat: 1
0 og 1 sammenlignes – resultat: 1
1 og 1 sammenlignes – resultat: 0

Resultatet er altså kun 1, hvis den ene af de to sammenlignede bit er 1, og den anden er 0. Eksempel:

Register A	1	1	0	1	0	1	0	1	= 213
På adresse HL	0	1	1	1	0	0	0	1	= 113
XOR (HL)	1	0	1	0	0	1	0	0	= 164

Læg mærke til, at det ikke er HL, der indeholder 113, men derimod indholdet på adressen specificeret i HL.

Der er disse muligheder med XOR:

Hex-kode	**Mnemonic**
AF	XOR A
A8	XOR B
A9	XOR C
AA	XOR D
AB	XOR E
AC	XOR H
AD	XOR L
AE	XOR (HL)
DDAE	XOR (IX + d)
FDAE	XOR (IY + d)
EEn	XOR n

Flagene sættes som for AND og OR.

XOR bruges bl.a. til at invertere (vende) bit, d.v.s. at give den enkelte bit den modsatte værdi. For eksempel kan vi ændre bit 5 til den modsatte værdi med instruktionen XOR 00100000 (XOR 32).

Bogstaverne i Amstrad CPC-464 er opbygget på en sådan måde, at koderne for de små bogstaver er lig koderne for de store – blot er der forskel på bit 5. Et »z« har således koden 122, og »Z« har koden 90:

z = 122 = 01111010
Z = 90 = 01011010

For at komme fra det ene bogstav til det andet, skal bit 5 blot inverteres. Vi prøver det i dette lille eksempel:

Hex-kode	**Mnemonic**
3E7A	LD A,122
CD5ABB	CALL TXT OUTPUT
EE20	XOR 32
CD5ABB	CALL TXT OUTPUT
C9	RET

A LOAD'es med koden for »z«. Dette udskrives, og bit 5 inverteres, hvorefter A indeholder koden for »Z«, der udskrives. Dernæst returneres til BASIC.

Læg i øvrigt mærke til, at en XOR A har samme virkning på A-registret som instruktionen LD A,0 – nemlig den, at A får værdien nul.

Register A	1	0	1	0	1	0	1	0	= 170
Register A	1	0	1	0	1	0	1	0	= 170
XOR A	0	0	0	0	0	0	0	0	= 0

Fordi XOR A er kortere end LD A,0, bruger man ofte denne instruktion til nulstilling af A-registret. Man må dog lige huske på, at carry-flaget samtidig resættes.

Prøv selv at konstruere et program, der bruger en eller flere af de logiske instruktioner. Herved kan du allerede nu blive fortrolig med ikke blot denne gruppe ordrer, men med hele maskinkodeprogrammeringen. Det er vigtigt, at du undervejs således udvikler små, forenklede og måske enfoldige programmer, som giver dig øvelse.

Kapitel 5

Vi skifter nu igen emne og går over til igen at betragte registrene som bytes, d.v.s indeholdende et helt tal og ikke blot en række bits.

Vi kan nemlig også i maskinkode lægge til og trække fra. Vi skal først vise brugen af instruktioner, der kun ændrer én ved et registers værdi, og derefter viser vi, hvordan man kan hhv. trække større tal fra og lægge større til.

De to instruktioner, der hhv. lægger én til og trækker én fra, hedder INC (INCrement) og DEC (DECrement). Dette kan ske på et 8 bits-register, på et 16 bits-register eller på indholdet på den adresse, HL-registret peger på (evt. også på (IX + d) og (IY + d)).

Vi tager et hurtigt eksempel:

Hex-kode	**Mnemonic**
3E41	LD A,65
214A9C	LD HL,40010
77	LD (HL),A
23	INC HL
3C	INC A
77	LD (HL),A
C9	RET

CALL 40000. Prøv derefter at skrive

```
PRINT PEEK 40010, PEEK 40011
```

Her skulle svarene gerne være 65 og 66. Hvorfor? Først LOAD'es A-registret med 65 og HL får værdien 40010. Indholdet i A lægges på adresse HL (altså lægges værdien 65 på adresse 40010); begge tællere, HL og A, incrementeres. Når der næste gang bruges ordren LD (HL),A, vil A indeholde værdien 66 (65 + 1), og den

værdi vil blive lagt på adresse 40011 (40010 + 1). Der returneres derefter.

DEC-instruktionen fungerer på samme måde, blot trækkes der én fra. Vi bringer her endnu et eksempel; dette viser funktionen af DEC (HL).

Hex-kode	**Mnemonic**
3E41	LD A,65
21489C	LD HL,40008
77	LD (HL),A
35	DEC (HL)
C9	RET

Med CALL 40000 : PRINT PEEK (40008) fås resultatet 64. Det skyldes, at vi først lægger 65 på adresse 40008, og derefter decrementerer indholdet på adresse HL – det er stadig 40008.

Her ses alle INC- og DEC-instruktionerne:

Hex-kode	**Mnemonic**	**Hex-kode**	**Mnemonic**
3C	INC A	3D	DEC A
04	INC B	05	DEC B
0C	INC C	0D	DEC C
14	INC D	15	DEC D
1C	INC E	1D	DEC E
24	INC H	25	DEC H
2C	INC L	2D	DEC L
34	INC (HL)	35	DEC (HL)
DD34d	INC (IX + d)	DD35d	DEC (IX + d)
FD34d	INC (IY + d)	FD35d	DEC (IY + d)
03	INC BC	0B	DEC BC
13	INC DE	1B	DEC DE
23	INC HL	2B	DEC HL
DD23	INC IX	DD2B	DEC IX
FD23	INC IY	FD2B	DEC IY

Man skal selvfølgelig bruge en instruktion som INC (IX + d) på samme måde som INC (HL), blot skal der som sædvanlig anvises en afstand fra IX's grundværdi. Hvis IX indeholder 40000, og vi bruger

INC (IX + 8)

så vil indholdet på adresse 40008 blive én større.

Hvad nu, hvis et 8 bits-register indeholder 255, og der så incrementeres. Resultatet kan vel ikke blive 256? Nej, selvfølgelig ikke – sådan en skrækkelig tanke må aldrig falde en maskinkodeprogrammør ind.

Vi kan se det i et eksempel:

Hex-kode	**Mnemonic**
3EFF	LD A,255
21489C	LD HL,40008
3C	INC A
77	LD (HL),A
C9	RET

Prøv så at kalde rutinen og finde værdien på adresse 40008. Hvad blev den? Nul? Hvis den gjorde det, så er din rutine rigtig indtastet og udført. Der sker nemlig det, at når et register har nået 255 (11111111 binært), så vil det, såfremt der lægges værdier til, begynde forfra med nul.

Tilbage står at fortælle, hvordan flagene påvirkes af INC og DEC. Her skal deles op i to grupper:

1) Er der tale om 8 bits-INC/DEC (incl. på en adresse gemt i et register – dette er jo også kun 8 bits, der regnes på), så vil der ske følgende:

   Z-flaget vil blive sat, hvis resultatet medfører nul
   S-flaget vil kopiere fortegnsbitten i resultatet
   P/V-flaget vil virke som overflowindikator, hvilket vi beskriver senere i dette kapitel
   C-flaget vil ikke blive berørt.

   Bemærk, at selv om et resultat overstiger 255 – som i eksemplet herover – så påvirkes C-flaget ikke.

2) Er der tale om 16 bits-INC/DEC, så vil ingen flag blive påvirket overhovedet.

Imidlertid vil det oftest være hensigtsmæssigt at skulle addere mere end én eller subtrahere mere end én. Skal vi f.eks. lægge 40 til A-registret, så vil instruktionen INC A 40 gange utvivlsomt være lidt triviel og besværlig. Heldigvis kan vi dog ty til andre metoder. Vi har fire ordrer til addition og subtraktion, og de hedder ADD (ADDition), SUB (SUBtraction), ADC (ADdition with Carry) og SBC (SuBtraction with Carry).

Her ses en liste over alle additions- og subtraktionsinstruktioner:

Hex-kode	**Mnemonic**	**Hex-kode**	**Mnemonic**
80	ADD A,B	88	ADC A,B
81	ADD A,C	89	ADC A,C
82	ADD A,D	8A	ADC A,D
83	ADD A,E	8B	ADC A,E
84	ADD A,H	8C	ADC A,H
85	ADD A,L	8D	ADC A,L
86	ADD A,(HL)	8E	ADC A,(HL)
87	ADD A,A	8F	ADC A,A
C6n	ADD A,n	CEn	ADC A,n
DD86d	ADD A,(IX+d)	DD8Ed	ADC A,(IX+d)
FD86d	ADD A,(IY+d)	FD8Ed	ADC A,(IY+d)
09	ADD HL,BC	ED4A	ADC HL,BC
19	ADD HL,DE	ED5A	ADC HL,DE
29	ADD HL,HL	ED6A	ADC HL,HL
DD09	ADD IX,BC		
FD09	ADD IY,BC		
DD19	ADD IX,DE		
FD19	ADD IY,DE		
DD29	ADD IX,IX		
FD29	ADD IY,IY		
90	SUB A,B	98	SBC A,B
91	SUB A,C	99	SBC A,C
92	SUB A,D	9A	SBC A,D
93	SUB A,E	9B	SBC A,E
94	SUB A,H	9C	SBC A,H
95	SUB A,L	9D	SBC A,L
96	SUB A,(HL)	9E	SBC A,(HL)
97	SUB A,A	9F	SBC A,A

D6n	SUB A,n	DEn	SBC A,n
DD96d	SUB A,(IX+d)	DD9Ed	SBC A,(IX+d)
FD96d	SUB A,(IY+d)	FD9Ed	SBC A,(IY+d)
		ED42	SBC HL,BC
		ED52	SBC HL,DE
		ED62	SBC HL,HL

Som det fremgår, så vil det altid være i A-registret, at en 8 bits-addition foregår. A vil derfor være den ene addend, og resultatet lægges altid i A. Når der er tale om en 16 bits-addition, så kan det foregå på HL, IX eller IY. Dog kan 16 bits-ADC kun fungere på HL-registret. Samtidig skal det nævnes, at der kun findes 16 bits-subtraktion på HL-registret, og da kun subtraktion med carry-flaget.

Vi udskyder ADC og SBC en omgang og ser først på almindelig addition og subtraktion.

Hex-kode	**Mnemonic**
3E32	LD A,50
1612	LD D,18
82	ADD A,D
CD5ABB	CALL TXT OUTPUT
06E6	LD B,230
80	ADD A,B
CD5ABB	CALL TXT OUTPUT
C9	RET

Vi LOAD'er A med 50 og adderer D's indhold – 18. A indeholder nu 68 (50 + 18), hvorefter karakteren med denne kode udskrives. Nu lægger vi 230 til de 68 i A-registret. Det giver 298. A-registret starter jo imidlertid forfra, når det når over 255, så værdien i A bliver 298 – 256 = 42. CHR$ (42) udskrives, inden der returneres. Programmet køres naturligvis med CALL 40000.

Vi tager et eksempel på subtraktion:

Hex-kode	**Mnemonic**
3E05	LD A,5
D607	SUB 7

```
CD5ABB      CALL TXT OUTPUT
C9          RET
```

CALL 40000.

A LOAD'es med 5, og der trækkes 7 fra. Dette giver – 2 tokomplementært, og det er jo 256 – 2 = 254. CHR$ (254) udskrives, hvorefter der returneres.

Når man bruger ADC, betyder det, at både værdien af carryflaget og værdien af det anvendte register lægges til; det vil sige 1, hvis flaget er sat, ellers 0. Vi kan se det i et eksempel:

Vi ønsker at addere DE og BC – og endda gemme resultatet i HL. Vi har ikke umiddelbart en sådan ordre, og vi kan derfor bruge ADC-instruktionen.

Vi lader BC indeholde 1000 og DE 1867 og betragter situationen binært:

```
Mente                 1 1 1 1   1       1
BC (1000)  0 0 0 0 0 0 1 1   1 1 1 0 1 0 0 0
DE (1867)  0 0 0 0 0 1 1 1   0 1 0 0 1 0 1 1
-------------------------------------------------
HL (2867)  0 0 0 0 1 0 1 1   0 0 1 1 0 0 1 1
```

Vi udfører additionen således:

Hex-kode	**Mnemonic**
79	LD A,C
83	ADD A,E
6F	LD L,A
78	LD A,B
8A	ADC A,D
67	LD H,A

Først lægges C over i A-registret og E adderes. Resultatet lægges i L. Dernæst lægges værdien i B over i A-registret, D *og en evt. carry* adderes, og resultatet lægges i H. Hvorfor nu bruge ADC? Jo, for ved den første addition vil der være sat en mente, som det ses her ovenfor på illustrationen. Når vi adderer C og E, vil vi addere 232 og 75, og det giver 307. Imidlertid kan A jo kun rumme op til 255, så de første 256 af de 307 trækkes fra. Resten – 51 – lægges

nu i A, og for at markere, at tallet egentlig behøvede en 9. bit, sættes C-flaget. Denne mente fra første addition lægges så automatisk til ved den anden addition ved brug af ADC.

Systemet er det samme ved subtraktion.

Bemærk desuden, at man i mange tilfælde vil foretrække at bruge 16 bits-addition/subtraktion, hvis man har mulighed for det.

Når man benytter 16 bits-subtraktion, vil man kun kunne bruge SBC, da SUB ganske enkelt ikke findes. Da kan det være nødvendigt at resætte C-flaget først, og som vi allerede har set i forrige kapitel, gøres det med AND A.

Hex-kode	**Mnemonic**
010080	LD BC,32768
2101C0	LD HL,49153
A7	AND A
ED42	SBC HL,BC
224D9C	LD (40013),HL
C9	RET

BC og HL LOAD'es med hhv. 32768 og 49153. Carry-flaget slettes, og BC trækkes fra HL. Resultatet lægges på adresserne 40013 og 40014. Med

```
CALL 40000 : PRINT PEEK (40013) + 256 * PEEK (40014)
```

skulle du derfor gerne få 49153 – 32768 = 16385.

Nu mangler vi blot at se, hvordan additions- og subtraktionsinstruktionerne påvirker flagene.

1) Er der tale om 16 bits-ADD, så vil der ske følgende:
   C-flaget vil blive sat, hvis resultatet medfører brug af en 17. bit
   De øvrige flag påvirkes ikke.
2) Er der tale om 8 bits-addition/subtraktion, hvad enten vi indregner C-flagets værdi eller ej, eller om 16 bits-ADC eller 16 bits-SBC, så vil der ske følgende:
   C-flaget sættes, hvis resultatet bliver større end 255 og derved har behov for en 9. bit

Z-flaget sættes, hvis resultatet bliver nul
S-flaget kopierer fortegnsbitten i resultatet
P/V-flaget virker her som overflowindikator og sættes derfor ved overflow

Vi skal her gøre rede for, hvad det vil sige, at der er overflow, og vi husker derfor to-komplement-metoden.

Hvis to positive tal adderes, og resultatet bliver over 127, så vil bit 7 (sign bit) blive sat. Dette betyder to-komplementært, at tallet er negativt, og det er jo forkert. Se blot her:

1)	34	0	0	1	0	0	0	1	0
	104	0	1	1	0	1	0	0	0
	138	1	0	0	0	1	0	1	0

Tallet ser to-komplementært ud til at være negativt. Her sættes P/V-flaget som tegn på, at der er noget galt: der er sket et overflow.

Adderes to negative tal, kan der ske følgende:

2)	− 34	1	1	0	1	1	1	1	0
	+− 104	1	0	0	1	1	0	0	0
	− 138	0	1	1	1	0	1	1	0

Hov! Sign bit er resat, men tallet er jo negativt! Her sættes P/V-flaget også – der er jo sket en fejl.

Hvis der derimod er tale om to tal med forskelligt fortegn, der adderes, så vil der intet overflow ske. Se dette eksempel:

3)	34	0	0	1	0	0	0	1	0
	+− 104	1	0	0	1	1	0	0	0
	− 70	1	0	1	1	1	0	1	0

Tallet er negativt og sign bit sat. Alt i orden – og Z80 har ikke hejst overflow-flaget.

For fuldstændighedens skyld:

4)		104	0	1	1	0	1	0	0	0
	+−	34	1	1	0	1	1	1	1	0
		70	0	1	0	0	0	1	1	0

Atter er alt i orden.

Ved subtraktion er det omvendt. Her kan overflowindikatoren kun blive sat, hvis to tal med modsat fortegn subtraheres, og resultatet er forkert to-komplementært.

Summa summarum: P/V-flaget indicerer under disse additions- og subtraktionsordrer, om der er sket et overflow. Dette forekommer, når et resultat af to tal i to-komplement-metoden er forkert udtrykt to-komplementært.

Kapitel 6

Man har også i maskinkode brug for at springe i et program. Springet kan enten være betinget (afhængigt af et flag) eller ubetinget.

Til dette formål bruges PC-registret. Det kan ikke LOAD'es med værdier som de øvrige registre, men det er heller ikke nødvendigt. PC (Program Counter) holder styr på, hvilken adresse i maskinkodeprogrammet Z80-processoren er nået til. PC har endvidere den egenskab, at inden den næste instruktion i rækken udføres, vil det indeholde adressen på den næste igen.

Eksempel:

Adresse	**Mnemonic**
40000	LD A,2
40002	XOR B
. . .	

Når instruktionen før LD A,2 er udført, fortæller PC-registret, at næste instruktion (LD A,2) ligger på adresse 40000. Inden LD A,2 udføres, vil PC finde næste instruktion, som ligger på adresse 40002. Altså antager PC værdien 40002, når LD A,2 udføres. Det hele gentages naturligvis med XOR B og de øvrige instruktioner.

Instruktionen, hvormed man kan springe i et program, hedder JP (JumP). Hvis der i et program står instruktionen

```
JP 40000
```

vil programudførelsen simpelt hen fortsætte på adresse 40000.

Man kan også have brug for at hoppe på en vis betingelse. Dette vil være et flags status, der så afgør, hvorvidt der skal springes eller ej. Hvis vi f.eks. ønsker, at vort maskinkodeprogram skal fortsætte fra adresse 20000, hvis bit 0 i A-registret er sat, skal følgende rutine bruges:

```
BIT 0,A
JP NZ,20000
```

Der udføres først en BIT 0,A. I kapitel 4 lærte du, at Z-flaget vil blive resat, hvis bitten, der testes (her bit 0 i A) er sat. Derefter står instruktionen JP NZ,20000. Den betyder, at hvis Z-flaget er NZ (Non Zero = resat), vil der blive sprunget til adresse 20000. Er Z-flaget sat, vil springet blive ignoreret, og der fortsættes, som om intet var hændt.

På samme måde kan JP betinges af C-, S- og P/V-flaget, så ialt findes disse instruktioner:

Hex-kode	**Mnemonic**	**Forklaring**
C3nn	JP nn	udføres altid
C2nn	JP NZ,nn	udføres, hvis Z=0
CAnn	JP Z,nn	udføres, hvis Z=1
D2nn	JP NC,nn	udføres, hvis C=0
DAnn	JP C,nn	udføres, hvis C=1
E2nn	JP PO,nn	udføres, hvis P/V=0
EAnn	JP PE,nn	udføres, hvis P/V=1
F2nn	JP P,nn	udføres, hvis S=0
FAnn	JP M,nn	udføres, hvis S=1

nn er et tal mellem 0 og 65535. Alle JP-instruktioner vil fylde 3 bytes.

Der findes endnu 3 JP-instruktioner:

Hex-kode	**Mnemonic**
E9	JP (HL)
DDE9	JP (IX)
FDE9	JP (IY)

JP (HL) hopper til adressen svarende til værdien i HL – og det må vel egentlig betegnes som overraskende! Da HL i mnemonic'en er angivet i parentes, og JP skal bruge en 2 bytes-adresse, skulle man umiddelbart tro, at der ville blive hoppet til indholdet på HL + 256 gange indholdet på (HL + 1), men dette er ikke tilfældet!!! Der hoppes som sagt blot til indholdet i HL.

JP (IX) og JP (IY) virker naturligvis på samme måde. Ingen af JP-instruktionerne påvirker flagene.

JP-instruktionen har en nært beslægtet instruktion i JR (Jump Relative). Med denne instruktion kan man hoppe relativt, d.v.s. i forhold til det sted, hvor man er. Mulighederne for at springe ligger mellem 128 bytes tilbage og 127 bytes frem, hvilket er i fuld overensstemmelse med det tidligere omtalte to-komplement-system.

Ønsker man at springe 100 bytes frem, så vil hex-koden hedde JR 100. I appendix A kan du se, at dette vil blive 1864h, idet 64 er 100 i hexadecimal notation. JR – 6 vil tilsvarende have hex-koden 18FA, idet – 6 i to-komplement-systemet er 256 – 6 = 250; det er FA i hexadecimal notation.

I modsætning til JP kan JR kun gøres betinget af to flag, men det er de to vigtigste. Der findes disse instruktioner:

Hex-kode	**Mnemonic**	**Forklaring**
18d	JR d	udføres altid
20d	JR NZ,d	udføres, hvis Z = 0
28d	JR Z,d	udføres, hvis Z = 1
30d	JR NC,d	udføres, hvis C = 0
38d	JR C,d	udføres, hvis C = 1

d er et tal mellem –128 og 127.

Inden JR-instruktionen i et program udføres, vil PC som beskrevet pege på næste instruktion. I dette tilfælde vil det være 2 bytes længere fremme, da alle JR-instruktioner fylder 2 bytes. Når JR udføres, hvor der springes fremad, vil d angive hvor mange bytes efter JR-instruktionen, der skal springes over. Se denne tegning:

Adresse	**Mnemonic**	
40000	JR	} JR 100 fylder to bytes
40001	100	
40002		Her peger PC før JR udføres
...		
40102		Hertil springes der

Inden JR udføres, peger PC på 40002. Her hoppes 100 frem, og PC vil indeholde 40102, og den næste instruktion, der vil blive udført, vil derfor stå på denne adresse.

Instruktionen JR −10 vil se således ud:

Hex-kode	**Mnemonic**	
40000		
40001		Der hoppes hertil
40002		
. . .		
40009	JR	JR − 10 fylder to bytes
40010	− 10	PC peger herpå, før JR udføres
40011		

Der hoppes således til 40011 − 10 = 40001.

Vi laver nu et program, der udskriver hele karaktersættet fra karakter 32 til 255. Programmet ser således ud:

Hex-kode	**Mnemonic**
3E20	LD A,32
CD5ABB	CALL TXT OUTPUT
3C	INC A
20FA	JR NZ, −6
C9	RET

Først LOAD'es A med 1. karakter, der skal udskrives, og den har koden 32. Denne udskrives dernæst, og A incrementeres for at tage næste karakter. Hvis A bliver nul, er karakteren med koden 255 netop udskrevet, og der er ikke flere, der skal udskrives. INC A vil sætte Z-flaget, hvis A er blevet nul. Derfor skal der udskrives en ny karakter, hvis Z er resat. Der springes 6 tilbage, så næste instruktion ved spring vil være CALL TXT OUTPUT. Hvis Z er sat, er programmet færdigt, og der fortsættes ned til RET, der returnerer til BASIC.

Vi skal nu se på en afart af JR-instruktionen. Den hedder DJNZ (Decrement Jump Non-Zero). Den er faktisk sammensat af to instruktioner:

DEC B
JR NZ,d hvor d er et tal mellem – 128 og 127

Som det ses, virker DJNZ på den måde, at først decrementeres B-registret. Hvis B er forskellig fra nul, vil der blive sprunget i programmet; nemlig d bytes frem.

Hvis man udfører en DJNZ – 2 vil det blot være en tom løkke udført værdien af register B gange. Hvis register B indeholder 0 fra starten, vil løkken køre 256 gange, fordi B decrementeres før der undersøges, om der skal hoppes (husk, at 0 – 1 = 255).

Prøv dette program:

Hex-kode	**Mnemonic**
3E41	LD A,65
0628	LD B,40
CD5ABB	CALL TXT OUTPUT
10FB	DJNZ – 5
C9	RET

A LOAD'es med karakterkoden for bogstavet A, mens B LOAD'es med det antal gange, bogstavet skal udskrives. Når B er blevet talt ned til nul, vil DJNZ ikke længere hoppe 5 bytes tilbage, og der returneres.

Der vil selvfølgelig komme bedre eksempler på brugen af DJNZ efterhånden som de mere slagkraftige programmer dukker op.

Som det ses i programeksemplet, er hex-koden:

Hex-kode	**Mnemonic**	
10d	DJNZ d	hvor d er et tal mellem –128 og 127

Fordi DJNZ-instruktionen kun virker på B-registret, der jo er et 8 bits-register, kan en løkke kun gennemløbes 256 gange. Man vil imidlertid tit i et program have brug for flere gennemløb, og da kan flg. metode benyttes. Hvis der f.eks. ønskes 36000 gennemløb, LOAD'es DE med dette antal, og hele proceduren vil se således ud:

```
LD DE,36000
...
opgave
...
DEC DE
LD A,D
OR E
JR NZ, opgave
```

Rutinen sammenligner bittene i D og E ved hjælp af en OR-instruktion. En enkelt sat bit i enten D eller E vil give et resultat forskelligt fra nul i A-registret, da jo blot én af de sammenlignede bit ved OR skal være sat for at få en sat bit som resultat. Vi vil altså kun opnå et nul i A-registret, hvis DE virkelig er talt ned til nul. Hvis DE er forskellig fra nul, vil Z-flaget (ved OR-instruktionens udførelse) resættes, og JR NZ medfører et hop tilbage.

Hvorfor overhovedet bruge en OR-instruktion til at finde ud af, om DE er blevet nul? Forklaringen er, at kun 8 bits-DEC påvirker Z-flaget.

Hverken JR eller DJNZ påvirker flagene. Det kan undre med hensyn til DJNZ, hvor man burde tro, at Z-flaget ville sættes, når B var talt ned til nul – men ingen regler uden undtagelse.

Kapitel 7

I maskinkode har man tit brug for at sammenligne registre og derved afgøre, om deres værdier er lig hinanden eller hvilken der er størst. Til dette formål bruges instruktionen CP (ComPare), som sammenligner indholdet i register A med indholdet i et 8 bits-register (A, B, C, D, E, H eller L) eller indholdet på en adresse ((HL), (IX + d) eller (IY + d)) eller et 8 bits-tal (mellem 0 og 255).

CP-instruktionen påvirker Z-flaget og C-flaget på en helt speciel måde. Hvis eksempelvis instruktionen CP B udføres (A sammenlignes med B), vil flagene påvirkes således:

1) hvis register A = register B, vil Z = 1 og C = 0
2) hvis register A > register B, vil Z = 0 og C = 0
3) hvis register A < register B, vil Z = 0 og C = 1

Hverken indholdet i register A eller B ændres.

Vi tager et lille eksempel:

Hex-kode	**Mnemonic**
3E41	LD A,65
CD5ABB	CALL TXT OUTPUT
3C	INC A
FE5B	CP 91
20F8	JR NZ, – 8
C9	RET

Dette program vil udskrive alle de store bogstaver, idet de har karakterkoderne fra 65 (A) til 90 (Z). Vi bruger CP-instruktionen til at finde ud af, om næste karakter, der står til at blive udskrevet, er CHR $ (91). Hvis vi ikke er nået til nr 91, vil Z-flaget være resat efter CP-instruktionen, og der hoppes tilbage i programmet. Ellers returneres der.

Vi vil nu lave et lidt større programeksempel, der også gør

brug af CP. Som du allerede har set, så har store bogstaver resat bit 5, hvor de små har den sat. Det vil vi benytte i den kommende rutine, der også gør flittigt brug af JR-instruktionen, som du lærte at kende i sidste kapitel.

Inden vi starter på selve programmeringen, skal vi stifte bekendtskab med endnu en af producenternes rutiner. Den hedder TXT RD CHAR, og vi vil behandle den dybere i kapitel 15. Med den kan vi aflæse en plads på skærmen og finde ud af, hvilken karakter der står dér. A-registret vil, når rutinen returnerer, indeholde karakterkoden på den fundne karakter. Den plads, der aflæses, er den nuværende markørposition.

Programmet, som vi vil lave, skal ændre skærmens små karakterer til store karakterer.

Vi starter med at bruge én af Amstrad's 32 indbyggede kontrolkoder, som du kan iagttage i appendiks E. Med kontrolkoden 30 vil markørens position blive flyttet til øverste venstre hjørne. Kontrolkoden udføres – ligesom udskrivning af karakterer – med rutinen TXT OUTPUT. Vi får altså

```
LD A,30
CALL TXT OUTPUT
```

Skærmen indeholder 1000 karakterer i mode 1 – som vi forventer, at du arbejder i – så vi får DE til at holde øje med, hvor mange karakterer der er tilbage. Startværdien for DE bliver altså 1000.

```
LD DE,1000
```

Bemærk, at i stedet for at tælle op fra 0 til 1000, så tæller vi *ned* fra 1000 til 0, da vi derved kan bruge Z-flaget til at finde ud af, om der er flere karakterer tilbage.

Så kalder vi TXT RD CHAR, der returnerer med den fundne karakter i A-registret. Første gang vil der blive undersøgt i skærmens øverste venstre hjørne, fordi kontrolkode 30 er blevet udført. Hvis karakteren er et lille bogstav, vil det have en karakterkode mellem 97 og 122. Hvis der ikke er tale om et lille bogstav, skal sekvensen med ændringen til stort bogstav springes over. Derfor laves denne rutine:

```
CP 97
JR C,11
CP 123
JR NC,7
```

Hvis A indeholder under 97 (97 > A), vil C-flaget blive sat, og der skal springes væk. Dernæst forestår en undersøgelse af, om A er under 123, for så vil der være tale om en lille karakter. Hvis A er 123 eller derover (123 < = A), vil C-flaget ved CP 123 blive resat, og der skal springes.

Såfremt der ikke er sprunget, vil vi altså have karakterkoden for et lille bogstav i A. Dennes bit 5 skal nu resættes for at få den konverteret til et stort bogstav. Karakterkoden for det store bogstav skal så udføres, og det gøres med CALL TXT OUTPUT.

```
RES 5,A
CALL TXT OUTPUT
JR 5
```

Når det store bogstav er udskrevet – dér, hvor det lille bogstav stod – vil TXT OUTPUT-rutinen selv rykke markørens position en plads længere til højre, så den er klar til næste karakter. Imidlertid vil dette jo ikke ske, hvis karakteren ikke skulle ændres (hvis der ikke var et lille bogstav), så vi springer 5 bytes frem. På disse 5 bytes står de to instruktioner, der sikrer, at markørens position vil blive rykket én til højre, hvis der ikke var tale om et lille bogstav.

```
LD A,9
CALL TXT OUTPUT
```

Kontrolkoden 9 vil sikre dette – se evt. appendiks E. Nu mangler vi blot at tælle DE ned, men som du allerede ved, så vil DEC DE ikke berøre Z-flaget. Derfor bruger vi

```
DEC DE
LD A,E
OR D
JR NZ, – 28
```

Hvis DE er blevet nul, er alle karakterer på skærmen efterprøvet, og der skal returneres.

```
RET
```

I sin helhed ser vort program således ud:

Hex-kode	**Mnemonic**
3E1E	LD A,30
CD5ABB	CALL TXT OUTPUT
11E803	LD DE,1000
CD60BB	CALL TXT RD CHAR
FE 61	CP 97
380B	JR C,11
FE7B	CP 123
3007	JR NC,7
CBAF	RES 5,A
CD5ABB	CALL TXT OUTPUT
1805	JR 5
3E09	LD A,9
CD5ABB	CALL TXT OUTPUT
1B	DEC DE
7B	LD A,E
B2	OR D
20E4	JR NZ, – 28
C9	RET

Der findes, ud over CP-instruktionen, fire andre sammenlignings-instruktioner, som er i familie med CP. Det er nogle specielle bloksøgningsinstruktioner, hvormed man kan søge i hele blokke af data efter en bestemt værdi. Den første hedder CPI (ComPare Increment), og denne instruktion udfører i sig selv disse 3 instruktioner:

```
CP (HL)
INC HL
DEC BC
```

Indholdet i (HL) sammenlignes med register A, og HL incrementeres, mens BC decrementeres. Hvis A og (HL) er ens, vil Z-flaget blive sat, i modsat fald resat. Vi kan desværre ikke få at vide, om A er større eller mindre end (HL), hvis Z-flaget resættes. C-flaget berøres nemlig ikke! S-flaget derimod kopierer på sædvanlig vis sign bit, mens P/V-flaget resættes, hvis BC er talt ned til nul.

Hvad nu, hvis der skal undersøges flere bytes? Til dette formål findes en tilsvarende instruktion, der blot repeterer CPI. Denne hedder CPIR (ComPare, Increment and Repeat). Når CPIR er afsluttet, kan der være to grunde. Den ene grund kan være, at den søgte værdi (som ligger i A-registret) er fundet. I så fald vil Z-flaget være sat, og værdien står på adresse (HL – 1), da HL jo incrementeres *efter* sammenligningen. Den anden grund er, at den søgte værdi ikke er fundet i blokken. BC er blevet talt ned til nul, da CPIR jo var repeterende. At den søgte værdi ikke er fundet afsløres af Z-flaget, som vil være resat.

Der findes også instruktionen CPD (ComPare Decrement), som udfører

```
CP (HL)
DEC HL
DEC BC
```

og som sætter flagene på samme måde som CPI. Den eneste forskel er altså blot, at HL efter sammenligningen decrementeres i stedet for at blive incrementeret.

Den fjerde instruktion af denne type hedder CPDR (ComPare, Decrement and Repeat), og den repeterer naturligvis CPD til værdien er fundet eller BC er blevet nul. Flagene påvirkes på samme måde.

Der gives ialt disse sammenligningsinstruktioner:

Hex-kode	**Mnemonic**
B8	CP B
B9	CP C
BA	CP D
BB	CP E
BC	CP C
BD	CP L

BE	CP (HL)
BF	CP A
FEn	CP n
DDBEd	CP (IX + d)
FDBEd	CP (IY + d)
EDA1	CPI
EDB1	CPIR
EDA9	CPD
EDB9	CPDR

Resten af kapitlet har vi helliget små instruktioner, der ikke har noget decideret tilholdssted.

Den første hedder EX DE,HL, og den er den eneste af sin slags. Når den udføres, vil værdierne i DE og HL ganske enkelt blive ombyttet. Det kan ofte være nyttigt i større programmer, da HL jo er et mere begunstiget 16 bits-register end DE.

En anden lille instruktion hedder NOP (No OPeration), og den udfører absolut intet! Det, man kan bruge den til, er, at hvis nogle bytes midt i et maskinkodeprogram bliver uønskede, så kan man indsætte NOP-instruktioner i stedet for. Selv om NOP ikke udfører noget, så tager det alligevel tid at eksekvere den. Det drejer sig dog om 1/1000000 af et sekund, hvilket du kan læse mere om i appendiks D. Hex-koden er meget passende 00h.

Der findes endvidere to instruktioner, som vedrører carry-flaget. De hedder SCF og CCF. SCF (Set Carry Flag) sætter C-flaget, mens CCF (Complement Carry Flag) giver C-flaget den modsatte værdi. Er det sat, bliver det resat ved udførelsen af CCF, og vice versa.

Den næste instruktion hedder CPL (ComPLement), og den vender hver eneste bit i register A. Eksempel:

Register A	1 1 0 1 0 0 1 0
CPL udføres	0 0 1 0 1 1 0 1

Dette kaldes også for et-komplementet.

En lignende instruktion hedder NEG (NEGate). Den giver register A værdien 256 – A, hvilket svarer til, at A får vendt sine bits og derefter tillagt én. Dette kaldes, som du allerede har set, to-

komplementet. Hvis A således indeholder 37, vil NEG medføre, at A's indhold ændres til 219 (– 37).

Hex-koderne for disse små instruktioner ser således ud:

Hex-kode	**Mnemonic**
EB	EX DE,HL
00	NOP
27	SCF
3F	CCF
2F	CPL
ED44	NEG

Kapitel 8

Der findes endnu et 16 bits-register i Z80-processoren, som vi ikke har beskrevet, men blot nævnt i oversigten i kapitel 3. Det er SP-registret (Stack Pointer). Det er et decideret 16 bits-register, hvilket jo betyder, at det ikke kan deles op i to 8 bits-registre.

SP peger på toppen af den såkaldte stak (deraf registrets navn), som er beliggende et sted i RAM. På stakken kan man gemme indholdet af diverse registre. Stakken fungerer således som et lager for værdier. Når indholdet af et register gemmes på stakken – med PUSH-instruktionen – lægges værdien øverst på stakken oven på dens øvrige værdier. Når man så ønsker at få en lagret værdi tilbage – ved hjælp af POP-instruktionen – så fås altid den øverste værdi på stakken. Det er dette sted, som SP peger på.

Bunden af stakken, dér, hvor den først lagrede værdi ligger, ligger højere oppe i RAM end toppen af stakken, hvor den sidst gemte værdi er placeret. Det betyder, at efterhånden som der kommer flere og flere tal op på stakken, bliver de placeret længere og længere nede i RAM adressemæssigt.

Vi tager et eksempel. Hvis staktoppen ligger på adresse 40000 (SP = 40000), vil det næste tal man gemmer (som altid er et 16 bits-tal) lægges på adresse 39998 og 39999, naturligvis med MSB i 39999.

Som sagt hedder instruktionen til lagring PUSH, og der findes disse muligheder:

Hex-kode	**Mnemonic**
C5	PUSH BC
D5	PUSH DE
E5	PUSH HL
F5	PUSH AF
DDE5	PUSH IX
FDE5	PUSH IY

Som allerede nævnt er det altid 16 bits-tal, der lagres.

Vi tager et lidt mere konkret eksempel. Hvis SP indeholder 40000 og BC 12345 (256 * 48 + 57), så vil PUSH BC forløbe således:

STAK:	Adresse		
	40000	←	Her peger SP før PUSH BC
B = 48	39999		
C = 57	39998	←	Her peger SP efter PUSH BC
	39997		

SP vil først decrementeres (her til 39999), hvorefter MSB af BC-registret, altså B-registret, vil lægges på den adresse, SP peger på (her 39999). Så decrementeres SP endnu en gang og LSB af BC vil lægges på adressen, SP nu peger på (39998).

Der forløber faktisk denne lille rutine:

```
DEC SP
LD (SP),B
DEC SP
LD (SP),C
```

Du skal lige bemærke, at LD (SP),B og LD (SP),C faktisk ikke findes i Z80-sproget. Vi har blot medtaget dem for at gøre PUSH-instruktionen mere klar for dig.

Når man PUSH'er en værdi op på stakken, vil værdien ikke forsvinde fra 16 bits-registret. Værdien på stakken er således blot en kopi.

Hvis vi udfører

```
PUSH BC
PUSH DE
PUSH HL
```

vil det forløbe således, hvis SP før instruktionernes udførelse indeholder 40000:

STAK:	Adresse		
	40000	←	SP peger her fra start
B	39999		
C	39998		
D	39997		
E	39996		
H	39995		
L	39994	←	Her peger SP efter rutinen
	39993		

HL-registret, der blev PUSH'et sidst, ligger som forventet på de to nederste pladser i stakken *adressemæssigt*, men stadig øverst på stakken!

Hvis IX indeholdende 30000 PUSH'es, sker der selvfølgelig dette:

STAK:	Adresse		
	40000	←	SP før
117	39999		
48	39998	←	SP efter
	39997		

MSB af IX ligger øverst adressemæssigt, og LSB nederst. 30000 = 256 * 117 + 48, så alt stemmer.

PUSH AF kræver måske en nøjere forklaring, da A og F jo ikke kan sammensættes til et 16 bits-register. Men PUSH-instruktionen kræver jo et sådan, så man lader bare som om, at AF er sat sammen. Det hele er ganske simpelt. PUSH AF forløber således:

STAK	Adresse		
	40000	←	SP før
A	39999		
F	39998	◄	SP efter
	39997		

Register A betragtes altså som MSB af de to registre.

Vi har også brug for at hente værdierne tilbage til registrene igen, og som sagt bruges instruktionen POP:

Hex-kode	**Mnemonic**
C1	POP BC
D1	POP DE
E1	POP HL
F1	POP AF
DDE1	POP IX
FDE1	POP IY

POP virker fuldstændig modsat af PUSH og udfører derfor egentlig disse fire instruktioner (POP HL benyttes):

```
LD L,(SP)
INC SP
LD H,(SP)
INC SP
```

Her skal det også understreges, at LD L,(SP), og LD H,(SP) ikke findes, men igen blot bruges til at overskueliggøre POP-instruktionen. Selvfølgelig virker POP også som PUSH kun med 16-bits registre.

Vi tager et eksempel. Stakken ser således ud:

STAK:	Adresse	
	30000	
1	29999	
2	29998	
3	29997	
4	29996	
5	29995	
6	29994	
7	29993	
8	29992	← SP peger på 29992
	29991	

Vi udfører disse instruktioner:

```
POP HL
POP DE
POP BC
```

Da vil HL indholde 256 * 7 + 8 = 1800, DE vil indeholde 256 * 5 + 6 = 1286, og BC vil indeholde 256 * 3 + 4 = 772. Stakken vil derefter se således ud:

STAK:	Adresse		
	30000		
1	29999		
2	29998	←	SP peger på 29998
3	29997		
4	29996		
5	29995		
6	29994		
7	29993		
8	29992		
	29991		

Bid mærke i, at de POP'ede tal stadig ligger i RAM, men SP peger nu på den nye staktop. Hvis der herefter ville blive udført POP HL, ville HL indeholde 256 * 1 + 2 = 258.

Se dette eksempel:

Hex-kode	**Mnemonic**
21409C	LD HL,40000
E5	PUSH HL
C1	POP BC
ED434A9C	LD (40010),BC
C9	RET

CALL 40000. Værdien i HL (40000) lægges op på stakken, hvorefter den hentes ned i BC. Herefter lægges værdien i BC på 40010 og 40011. Det betyder, at

```
PRINT PEEK (40010) + 256 * PEEK (40011)
```

skulle give svaret 40000.

Hvis vi ønsker at ombytte indholdet i register BC og DE bruges flg. fremgangsmåde:

```
PUSH DE
PUSH BC
POP DE
POP BC
```

Forklar det for dig selv.

Hvis vi ønsker at ombytte regiser D og E *uden* at berøre de øvrige registre, må vi gøre således:

```
PUSH DE
PUSH DE
INC SP
POP DE
INC SP
```

Efter at vi har PUSH'et to gange, ser stakken således ud:

STAK:	Adresse		
	30000	←	Her peger SP før start
D	29999		
E	29998		
D	29997		
E	29996	←	SP peger nu på 29996
	29995		

SP incrementeres (en helt ny instruktion), og stakken ser nu således ud:

STAK:	Adresse		
	30000		
D	29999		
E	29998		
D	29997	←	SP peger nu på 29997
E	29996		

Nu udføres POP DE, og ifølge vor beskrivelse af denne instruktion, vil E nu indeholde værdien på adresse 29997 (D) og D værdien på adresse 29998 (E). Nu er D og E altså ombyttet, og stakken ser således ud:

STAK:	Adresse		
	30000		
D	29999	←	SP peger nu på 29999
E	29998		
D	29997		
E	29996		

Nu mangler SP bare at blive sat til de 30000, som den var i starten, og dette gøres med INC SP. Grunden til, at vi absolut skal incrementere Stack Pointer, er, at returadressen til BASIC (hvilket vi beskriver nærmere i kapitlet efter dette) ligger på stakken, og skaber vi uorden i dette system, vil vi returnere til et forkert sted, og computeren vil lave mærkelige ting – tit et totalt reset.

Derfor skal man altid i et program være omhyggelig med stakken. I et program uden manipulation af stakken skal der således være lige mange PUSH og POP-instruktioner.

Vi ser vor ombytning fra før i et konkret eksempel:

Hex-kode	**Mnemonic**
011027	LD BC, 10000
C5	PUSH BC
C5	PUSH BC
33	INC SP
C1	POP BC
33	INC SP
ED434D9C	LD (40013), BC
C9	RET

CALL 40000 : PRINT PEEK (40013) + 256 * PEEK (40014) vil give resultatet 4135. Før indeholdt BC 10000 = 256 * 39 + 16, men efter ombytningen vil det indeholde 256 * 16 + 39 = 4135.

INC SP og de øvrige instruktioner, som virker på SP-registret, er opført herunder:

Hex-kode	**Mnemonic**
33	INC SP
3B	DEC SP
31nn	LD SP,nn
F9	LD SP,HL

ED7Bnn	LD SP,(nn)
ED73nn	LD (nn),SP
39	ADD HL,SP
ED7A	ADC HL,SP
ED72	SBC HL,SP
DD39	ADD IX,SP
FD39	ADD IY,SP
E3	EX (SP),HL
DDE3	EX (SP),IX
FDE3	EX (SP),IY

Med LD SP,nn kan man oprette flere stakke i RAM, hvis det behøves. Man skal dog altid være opmærksom på returadresser – i det hele taget er stak-fejl en hyppig fejlkilde i et program.

LD SP,HL kopierer HL-registret i SP.

LD SP,(nn) giver SP-registret værdien på en adresse nn + 256 * værdien på adresse (nn + 1).

LD (nn),SP lægger omvendt værdien af Stack Pointer på to adresser.

EX (SP),HL er endnu en EXchange-instruktion, og den ombytter værdien øverst på stakken med HL. Hvis HL indeholder 30000 = 256 * 117 + 48, og der på staktoppen ligger 20000 = 256 * 78 + 32, er situationen således:

STAK:	Adresse		
	30000		
78	29999		
32	29998	←	SP peger her – HL = 30000

Efter EX (SP),HL ser det således ud:

STAK:	Adresse		
	30000		
117	29999		
48	29998	←	SP peger stadig her – HL = 20000

Øverst på stakken står nu de 30000, som HL indeholdt før, og SP's værdi er ikke blevet ændret.

Der findes ingen EX (SP),BC. Hvis man måtte ønske at bruge en sådan instruktion, så bliver man nødt til at lave sin egen lille rutine, der klarer jobbet:

```
EX (SP),HL
PUSH BC
EX (SP),HL
POP BC
EX (SP),HL
```

Selvfølgelig benyttes samme fremgangsmåde for de andre registre, f.eks. DE-registret.

PUSH og POP berører ikke flagene, dog med en undtagelse, nemlig POP AF. F-registret modtager naturligvis sin værdi fra stakken. Hvis denne ser således ud:

STAK:	Adresse	
	30000	
100	29999	
170	29998	← SP peger her

Vi udfører dernæst POP AF, og da vil register A indeholde 100, og flag-registret vil indeholde de 170. Omskrevet til binær notation ser det således ud:

S	Z		H		P/V	N	C
1	0	1	0	1	0	1	0

10101010 i binær notation er 170

På denne måde kan man altså LOAD'e flag-registret indirekte. Vi vil ikke bringe noget større eksempel på stakinstruktionerne, men i resten af bogen vil de blive benyttet så flittigt, at du snart vil blive fortrolig med dem.

Kapitel 9

I maskinkode kan man have brug for at kalde et underprogram, der ved dets afslutning returnerer til hovedprogrammet. Dette er specielt anvendeligt i programmer, hvor en del af en rutine skal udføres flere gange. Denne programstump kan så kaldes, hver gang der er behov for det.

Kaldet foregår med instruktionen CALL, som du jo allerede har stiftet bekendtskab med i forbindelse med Amstrad's indbyggede rutiner. Kaldet kan endvidere gøres afhængigt af et flags status. Der findes ialt disse instruktioner:

Hex-kode	**Mnemonic**
CDnn	CALL nn
C4nn	CALL NZ,nn
CCnn	CALL Z,nn
D4nn	CALL NC,nn
DCnn	CALL C,nn
E4nn	CALL PO,nn
ECnn	CALL PE,nn
F4nn	CALL P,nn
FCnn	CALL M,nn

Vi har allerede i forbindelse med JP-instruktionen set, hvordan betingelserne fungerer, men vi rekapitulerer dem her:

CALL NZ udføres, hvis Z-flaget er resat (Z = 0)
CALL Z udføres, hvis Z-flaget er sat (Z = 1)
CALL NC udføres, hvis C-flaget er resat (C = 0)
CALL C udføres, hvis C-flaget er sat (C = 1)
CALL PO udføres, hvis P/V-flaget er resat (P/V = 0)
CALL PE udføres, hvis P/V-flaget er sat (P/V = 1)
CALL P udføres, hvis S-flaget er resat (S = 0)
CALL M udføres, hvis S-flaget er sat (S = 1)

Der kaldes naturligvis en rutine på adresse nn (en 16 bits-adresse).

Når underrutinen er færdig, returneres med instruktionen RET (RETurn), som du allerede har set i forbindelse med returnering til BASIC. Denne instruktion kan også gøres betinget, så der returneres, hvis det pågældende flag har den rette status. Eksempel: RET NZ udføres kun, hvis Z-flaget er resat. Ialt findes disse instruktioner:

Hex-kode	**Mnemonic**
C9	RET
C0	RET NZ
C8	RET Z
D0	RET NC
D8	RET C
E0	RET PO
E8	RET PE
F0	RET P
F8	RET M

Hvortil returneres der, når RET udføres? Når vi kalder et underprogram, vil adressen på instruktionen efter CALL-instruktionen blive lagt op på stakken – og når der returneres, vil der blive vendt tilbage til adressen beliggende øverst på stakken. Man kan således sige, at ved CALL PUSH'es PC (programpilen), mens PC POP'es ved RET. Hvis vi betragter dette program:

Adresse	**Mnemonic**
40000	CALL 41000
40003	. . .

vil der ved udførelsen af CALL 41000 ske det, at tallet 40003 lægges op på stakken. Hvis denne før ser således ud:

STAK:	Adresse		
	42000	←	SP peger her
	41999		
	41998		
	41997		

vil den efter CALL-instruktionens udførelse være ændret til:

STAK:	Adresse		
	42000		
9Ch	41999		
43h	41998	←	SP peger her
	41997		

9C43 i hexadecimal notation er 40003. Efter udførelsen af CALL, fortsættes fra adresse 41000.

Når der returneres, vil der – forudsat, at der ingen ændringer i Stack Pointers værdi sker – hænde følgende med stakken:

STAK:	Adresse		
	42000	←	SP peger her
9Ch	41999		
43h	41998		
	41997		

PC indeholder nu 40003, hvorfra der fortsættes.

Det er derfor meget vigtigt at holde styr på stakken, da returneringer ellers formodentlig ikke vil forløbe tilfredsstillende. Der må således – som allerede nævnt i sidste kapitel – holdes et skarpt øje med antallet af PUSH- og POP-instruktioner. Husk også, at returadressen til BASIC også er lagret på stakken.

Imidlertid kan det lade sig gøre at manipulere med stakken og dens returadresser. Vi lærte i kapitel 6, at man med JP (HL) vil springe til adressen indeholdt i HL. Det samme kan gøres med IX og IY, men ikke med BC og DE. Det kan dog let arrangeres. JP (BC) kan således laves med:

```
PUSH BC
RET
```

Indholdet i BC kopieres på stakken, og når der udføres en RET, vil PC komme til at indeholde stakkens øverste værdi – i dette eksempel altså værdien i BC.

Vi kan også se systemet i dette eksempel:

Hex-kode	**Mnemonic**
214A9C	LD HL,40010
115ABB	LD DE,BB5Ah
3E07	LD A,7
E5	PUSH HL
D5	PUSH DE
C9	RET

Efter at HL og DE er PUSH'et, ser stakken således ud:

STAK:	
*	} Her ligger returadres-
*	} sen til BASIC
9C	MSB af 40010
4A	LSB af 40010
5A	MSB af BB5Ah
BB	LSB af BB5Ah – SP peger nu her

Nu udføres en RET, og derfor fortsætter programudførelsen i BB5Ah (det er TXT OUTPUT-rutinen). Nu ser stakken således ud:

STAK:	
*	} Returadressen til
*	} BASIC
9C	MSB af 40010
4A	LSB af 40010 – SP peger nu her
5A	
BB	

Når kontrolkode 7 (en lyd) er udført ved hjælp af den kaldte TXT OUTPUT-rutine, vil der i ROM stå en RET-instruktion. Når den mødes, vil stakken atter få lagt sin øverste værdi i PC. Der står 9C4Ah, så programudførelsen fortsætter derfra. På denne adresse (40010) står en RET, så der hentes en ny værdi på stakken – og det er returadressen til BASIC.

Det er således muligt at manipulere med stakken, men vi råder dig til at gøre det med forsigtighed.

Vi har nu forklaret, hvordan underprogrammer (normalt kaldet subrutiner) fungerer, men man behøver altså ikke blot kalde egne subrutiner. Vi har allerede set, at

```
CALL TXT OUTPUT
```

kalder en rutine i ROM, og når der dernede mødes en RET, returneres der til instruktionen efter CALL i vort program. Alle disse rutiner, som man kan have nytte af at kalde, gennemgår vi nøjere i kapitel 21.

Vi vil nu lave et program, der udskriver decimaltal. Fremgangsmåden bliver således: det pågældende tal ligger i HL. Det fratrækkes først 10000. Hvis resultatet bliver negativt, stoppes fratrækningen. Ellers trækkes yderligere 10000 fra, indtil tallet bliver negativt. En tæller holder styr på, hvor mange gange 10000 er trukket fra. Dette tal vil være 1. ciffer i decimaltallet. Dette vil blive udskrevet, og 10000 lægges til det opnåede negative tal, så dette tal bliver resten af det pågældende tal, efter at første ciffer er udskrevet. Derefter fratrækkes dette tal 1000 o.s.v.

Programmet, der trækker et tal fra, undersøger om det er negativt o.s.v., vil blive lagt i en underrutine. Når underrutinen kaldes, skal BC indeholde faktoren, der skal trækkes fra og HL tallet, som BC skal trækkes fra. Endvidere lader vi E-registret holde styr på, om der er udskrevet tal. Hvis HL f.eks. fra start indeholder 7890 og 10000 trækkes fra, vil tælleren angive nul, men dette skal jo ikke udskrives (udskriften skal ikke være 07890).

Vort første skridt i underprogrammet er at LOAD'e A med 47.

```
LD A,47
```

Vi undersøger nu, om der ved fratrækningen (første gang med 10000) opstår et negativt tal. Hvis dette ikke er tilfældet, skal tælleren øges med én, og vi skal trække fra på ny. Det giver dette:

```
INC A
AND A
```

```
SBC HL,BC
JR NC, – 6
```

Ved første gennemløb forøges A-registrets værdi til 48, og C-flaget resættes. Så trækkes BC (10000 ved første kald) fra det tal, der skal udskrives. Hvis dette er negativt, vil C-flaget være sat. Derfor springes til endnu en subtraktion, hvis C-flaget er resat. Hvis tallet f.eks. er 7890, vil C-flaget blive sat, og A indeholder nu 48, der er koden for 0. Hvis tallet havde været 17890, havde A indeholdt 49 – koden for 1, der er det første ciffer i tallet.

Nu skal vi undersøge, om tallet er forskelligt fra nul. Er dette tilfældet, skal det udskrives

```
CP 48
JR NZ,3
```

Hvis A er 48 (koden for nul) skal tallet kun udskrives, hvis der er andre cifre udskrevet, eller hvis det er det sidste ciffer! Vi skal senere se, at E-registret fra start er 48 og således vil blive ændret, når et tal udskrives samt når det er det sidste ciffer.

```
CP E
JR Z,4
```

Hvis E er 48, ligesom A er det, skal det ikke udskrives, og der hoppes væk,

Nu står vi imidlertid foran udskrivning af tallet, og det gøres med kald af TXT OUTPUT. Desuden skal E ændre værdi (nu sker der jo en udskrift og efterfølgende nul skal udskrives), og vi har valgt at incrementere E.

```
INC E
CALL TXT OUTPUT
```

HL indeholder stadig det negative tal. For at få de resterende cifre frem, skal BC atter tillægges HL. Hvis HL indeholder 7890, og der trækkes 10000 fra, indeholder HL – 2110 (63426). Nu lægges atter 10000 til for at få 7890 frem, inden der trækkes 1000 fra.

```
ADD HL,BC
```

Underprogrammet er hermed slut.

```
RET
```

Nu til hovedprogrammet. Først skal E initialiseres.

```
LD E,48
```

Så skal rutinen kaldes med BC indeholdende hhv. 10000, 1000, 100, 10 og 1.

```
LD BC, 10000
CALL SUB
LD BC, 1000
CALL SUB
LD BC, 100
CALL SUB
LD C, 10
CALL SUB
LD C, 1
```

Bemærk, at vi i de to sidste tilfælde kun LOAD'er C, idet B jo allerede indeholder nul. Dets værdi ændres ikke i underrutinen. Ved denne fremgangsmåde spares bytes.

Når C er LOAD'et med 1, skal subrutinen også kaldes, men hvis vi lægger denne i forlængelse af hovedprogrammet, kan vi spare en CALL-instruktion. RET-instruktionen i slutningen af underprogrammet vil således sikre returnering til BASIC.

Inden vi »glider over« i subrutinen, skal E incrementeres, for at tallet nul udskrives.

```
INC E
```

I sin helhed giver det dette program:

Hex-kode	**Mnemonic**
1E30	LD E,48

```
011027      LD BC,10000
CD5C9C      CALL SUB
01E803      LD BC,1000
CD5C9C      CALL SUB
016400      LD BC,100
CD5C9C      CALL SUB
0E0A        LD C,10
CD5C9C      CALL SUB
0E01        LD C,1
1C          INC E

SUB
3E2F        LD A,47
3C          INC A
A7          AND A
ED42        SBC HL,BC
30FA        JR NC, – 6
FE30        CP 48
2003        JR NZ,3
BB          CP E
2804        JR Z,4
1C          INC E
CD5ABB      CALL TXT OUTPUT
09          ADD HL, BC
C9          RET
```

Inden rutinen kaldes, skal HL således indeholde tallet, der udskrives. Programmet er placeret fra adresse 40000, hvilket betyder, at en evt. flytning skal medføre ændring i hex-koden for kaldet af underrutinen.

Bemærk i øvrigt, at vi har indført en ny notation. Underprogrammet har fået et navn, SUB, og dette står anført ved dets begyndelse – af pladsmæssige hensyn i rækken af hex-koder. Dette kaldes en etiket, og vi vil ofte benytte denne form for navngivning i det følgende.

Afprøv selv programmet.

Kapitel 10

Man kan få brug for at flytte hele blokke af data. Til dette brug har Z80 i sit instruktionssæt 4 forskellige ordrer. Den første hedder LDI (LoaD and Increment), og den udfører i sig selv denne lille rutine:

```
LD (DE),(HL)
INC DE
INC HL
DEC BC
```

Bemærk, at instruktionen LD (DE),(HL) ikke findes som enkeltinstruktion. Her er der altså tale om en fiktiv instruktion, som vi har medtaget for at gøre det mere overskueligt.

LDI kopierer indholdet på adresse HL på adresse DE, og samtidig decrementeres en tæller i BC. Desuden incrementeres både DE og HL, så de er klar til en ny overførsel.

P/V-flaget er det eneste flag, der berøres. Det bliver resat, hvis tælleren BC når ned til nul.

Vi kunne således kopiere 5 adressers indhold på 5 andre adresser med

```
LDI
LDI
LDI
LDI
LDI
```

Imidlertid ville det nok være lidt besværligt i længden, og derfor har vi heldigvis en ordre, der klarer det hele selv. Den hedder LDIR (LoaD, Increment and Repeat), og den repeterer LDI-instruktionen indtil BC bliver nul. Skal vi således have overført 100

bytes liggende på adresserne 42100, 42101, 42102 og op til 42199 til de 100 bytes fra 43100 til 43199, så kan det gøres så enkelt:

```
LD DE,43100
LD HL,42100
LD BC,100
LDIR
```

DE indeholder startadressen på stedet, hvor de anvendte bytes skal kopieres – HL startadressen på stedet, hvor de står i øjeblikket – og BC indeholder antallet af bytes, der skal flyttes.

Vi vil nu lave et program, der kan gemme skærmbilledet et sted i RAM for senere at kalde det tilbage. Skærmen ligger i RAM fra adresse 49152 til 65535. Vi vil i kapitel 15 vende tilbage til opbygningen af RAM og af skærmen, men i dette eksempel er det overflødigt at vide. Skærmen fylder således 16384 bytes, og derfor skal der altså i alt gemmes dette antal af bytes. Det svarer til 16 K af hukommelsen.

Vi lægger skærmbilledet, der skal gemmes, i 16384 og må derfor LOAD'e DE med dette tal. HL skal indeholde skærmbilledets første byte, og det er 49152. I alt skal der overføres 16384, så BC LOAD'es med dette tal.

Det giver dette program:

```
LD DE,16384
LD HL,49152
LD BC,16384
LDIR
RET
```

Når skærmbilledet lagres, sker der naturligvis intet med selve skærmen. Der er jo blot tale om en kopiering.

Når man så får lyst til at hente det tilbage, skal DE og HL bytte værdier, da det er de bytes, der ligger fra 16384 til 32767, der skal overføres til 49152 til 65535. Vi får altså dette program:

```
LD DE,49152
LD HL,16384
```

```
LD BC,16384
LDIR
RET
```

Vi kan nu sammensætte disse to dele i én stor rutine, og som du måske har bemærket, er der en passage i programmerne, der er ens. Vi får derfor denne rutine:

Hex-kode	**Mnemonic**
GEM	
110040	LD DE,16384
2100C0	LD HL,49152
1806	JR 6
KALD	
1100C0	LD DE,49152
210040	LD HL,16384
010040	LD BC,16384
EDB0	LDIR
C9	RET

Hvis rutinen GEM kaldes, vil den efter at have LOAD'et DE og HL gå ned og LOAD'e BC for derefter at udføre dataoverførelsen. Bemærk, at vi igen bruger etiketter.

Du kan kalde rutinen med CALL 40000, hvorved skærmbilledet lagres, og det kan genfremkaldes med CALL 40008. Hvis skærmen bliver scrollet mellem kaldene af GEM og KALD, vil der dog ske mærkelige ting. Disse forklarer vi i kapitel 15.

På tilsvarende vis som med LDI og LDIR findes LDD (LoaD and Decrement) og LDDR (LoaD, Decrement and Repeat). Disse to decrementerer blot DE og HL i stedet. LDD udfører således:

```
LD (DE),(HL)
DEC DE
DEC HL
DEC BC
```

LDDR repeterer LDD, til BC bliver nul.

Hex-koderne for de 4 instruktioner kan du se her:

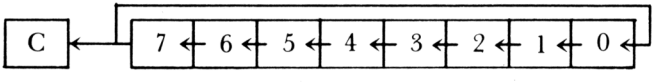
Hex-kode	**Mnemonic**
EDA0	LDI
EDB0	LDIR
EDA8	LDD
EDB8	LDDR

LDI og LDD sætter som nævnt P/V-flaget, hvis BC ikke er talt ned til nul, og heraf følger, at LDIR og LDDR altid resætter P/V-flaget.

Kapitel 11

I Z80-sproget findes nogle såkaldte rotations- og skifteinstruktioner. De flytter rundt på bittene (til højre eller til venstre) i et af 8 bits-registrene (A, B, C, D, E, H eller L) eller på en adresse ((HL), (IX + d) eller (IY + d)). Vi viser her, hvordan de virker, og vi gør det skematisk.

Den første instruktion hedder RLC (Rotate Left Carry), og den ser således ud:



Som det ses, vil alle bittene flytte én bit til venstre, mens bit 0 og carry-flaget vil få værdien af bit 7. Vi tager et eksempel:

Register A	1 0 0 1 1 0 1 0	Carry-flag = 0
RLC A udføres		
Register A	0 0 1 1 0 1 0 1	Carry-flag = 1

Som det er illustreret, er bit 7 (der er sat) rykket til bit 0 og C-flaget, der således nu er blevet sat. Vi kan vise RLC i dette eksempel:

Hex-kode	**Mnemonic**
3EAA	LD A,170
CD5ABB	CALL TXT OUTPUT
CB07	RLC A
CD5ABB	CALL TXT OUTPUT
C9	RET

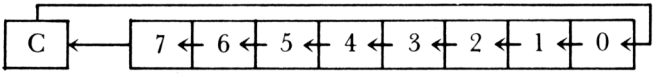
Efter CALL 40000 vil du få udskrevet CHR$ (170). Så roterer vi A-registret.

Register A før	1	0	1	0	1	0	1	0	= 170	Carry-flag = ?
Register A efter	0	1	0	1	0	1	0	1	= 85	Carry-flag = 1

Efter rotationen indeholder A-registret 85 og C-flaget er sat – det sidstnævnte bruger vi ikke i ovenstående program. CHR$ (85) udskrives ved kald af TXT OUTPUT.

De flg. instruktioner, der er analog til RLC, vil vi ikke give eksempler på, men vi råder dig til selv at eksperimentere.

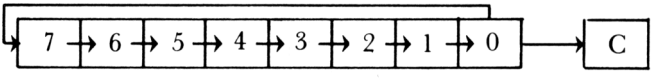
RL (Rotate Left):



Bit 7 går til carry-flaget, og carry-flagets værdi til bit 0. Denne gang er det C-flaget (og ikke bit 7), som rykker til bit 0. Eksempel:

Register A	0	1	0	1	0	1	0	0	Carry-flag = 1
RL A udføres									
Register A	1	0	1	0	1	0	0	1	Carry-flag = 0

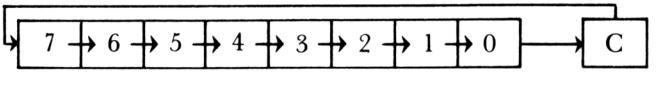
RRC (Rotate Right Carry):



Bit 0 går til bit 7 og carry-flaget. Eksempel:

Register B	1	0	1	0	1	0	1	0	Carry-flag = 1
RRC B udføres									
Register B	0	1	0	1	0	1	0	1	Carry-flag = 0

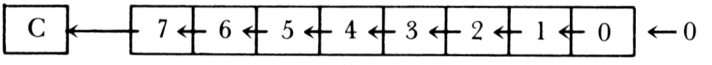
RR (Rotate Right):



Bit 0 går til carry-flaget, og C-flagets værdi til bit 7. Eksempel:

Register D 1 0 1 0 1 0 1 0 Carry-flag = 1
RR D udføres
Register D 1 1 0 1 0 1 0 1 Carry-flag = 0

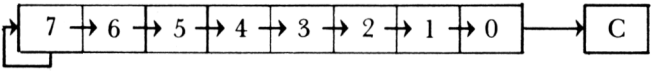
SLA (Shift Left Arithmetic):



Bit 7 går til C-flaget, og bit 0 resættes. Eksempel:

Register E 1 0 1 0 1 0 0 1 Carry-flag = 0
SLA E udføres
Register E 0 1 0 1 0 0 1 0 Carry-flag = 1

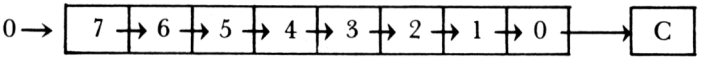
SRA (Shift Right Arithmetic):



Bit 7 forbliver uændret, og bit 0 går til carry-flaget. Eksempel:

Register E 1 0 0 1 0 0 1 1 Carry-flag = 0
SRA E udføres
Register E 1 1 0 0 1 0 0 1 Carry-flag = 1

SRL (Shift Right Logic):



Bit 0 går til carry-flaget, og bit 7 resættes. Eksempel:

Register H 1 0 0 1 0 0 1 1 Carry-flag = 1
SRA H udføres
Register H 0 1 0 0 1 0 0 1 Carry-flag = 1

I skemaet herunder står hex-koder for alle de mulige rotations- og skifteinstruktioner. Du skal imidlertid være lidt påpasselig, når du bruger skemaet, for du kan ikke bruge hex-koderne

umiddelbart. Koderne skal have CB foran sig, undtagen for IX og IY, hvor CB skal være den anden byte i hex-koden. Desuden skal d – distancen – være tredje byte.

	B	C	D	E	H	L	(HL)	A	(IX+d)	(IY+d)
RLC	00	01	02	03	04	05	06	07	DD06	FD06
RRC	08	09	0A	0B	0C	0D	0E	0F	DD0E	FD0E
RL	10	11	12	13	14	15	16	17	DD16	FD16
RR	18	19	1A	1B	1C	1D	1E	1F	DD1E	FD1E
SLA	20	21	22	23	24	25	26	27	DD26	FD26
SRA	28	29	2A	2B	2C	2D	2E	2F	DD2E	FD2E
SRL	38	39	3A	3B	3C	3D	3E	3F	DD3E	FD3F

Instruktionen RRC L har således hex-koden CB0D. SRL (HL) har hex-koden CB3E, og SRL (IX + 26) hedder DDCB1A3E, da 26 i hexadecimal notation er 1A. Bemærk som nævnt, at de 26 er tredje tal i hex-koden. Dette har vi også tidligere beskrevet under brugen af IX- og IY-registrene. SRA (IY – 100) har koden FDCB9C2E. Igen er – 100 klemt inde som tredje byte. – 100 er to-komplementært 9Ch.

Der findes yderligere 4 rotationsinstruktioner. De hedder:

Hex-kode	**Mnemonic**
07	RLCA
0F	RRCA
17	RLA
1F	RRA

Det mærkelige er, at RLCA med hex-kode 07 udfører nøjagtig det samme som RLC A, som har hex-koden CB07. Der spares således en byte. RRCA svarer naturligvis til RRC A, RLA til RL A og RRA til RR A.

Vi bruger nu rotationsinstruktionerne i et lille program, der skal udskrive et tal binært for os. Rutinen ser således ud:

Hex-kode	Mnemonic
21409C	LD HL,40000
0E0A	LD C,10
LØKKE 1	
7E	LD A,(HL)
0608	LD B,8
LØKKE 2	
07	RLCA
F5	PUSH AF
3E30	LD A,48
CE00	ADC A,0
CD5ABB	CALL TXT OUTPUT
F1	POP AF
10F4	DJNZ LØKKE 2
3E20	LD A, 32
CD5ABB	CALL TXT OUTPUT
CD5ABB	CALL TXT OUTPUT
23	INC HL
0D	DEC C
20E5	JR NZ, LØKKE 1
C9	RET

Rutinen udskriver som sagt binære tal. Vi har lavet det sådan, at ialt 10 tal udskrives (C indeholder dette tal). HL peger på 40000 fra start, for dér ligger de værdier, vi ønsker at udskrive binært.

A LOAD'es med værdien på den pågældende adresse, og B indeholder 8. Rutinen, der udskriver de binære tal (denne kaldes LØKKE 2), skal således gennemløbes 8 gange, en gang for hvert ciffer. A roteres så til venstre. Hvis C-flaget er resat, skal nul (karakterkode 48) udskrives, mens et sat C-flag skal give én (karakterkode 49) i udskrift. Instruktionen ADC A,0 bliver således anvendt for at tillægge A værdien af carry-flaget, og denne er 1, hvis udskriften skal være 1. Inden selve udskriften PUSH'es AF for at huske den værdi, der var i færd med at blive roteret, og den POP'es tilbage efter udskriften. Der tages så et nyt ciffer, hvis der ikke er udskrevet 8. Dernæst udskrives mellemrum.

HL incrementeres dernæst, så den peger på næste byte, og C decrementeres for at undersøge, om der er flere bytes. Denne ru-

tine kan sagtens bruges som underrutine i programmer, hvor binære tal skal udskrives.

Der findes endnu to rotations-instruktioner, men de har at gøre med BCD-aritmetik (Binary Coded Decimal), og det er en helt speciel form for aritmetik:

I 4 bit kan man udtrykke ialt 2 * 2 * 2 * 2 = 16 forskellige værdier. I det hexadecimale system kaldes disse værdier eller tal om man vil for 00, 01, 02 o.s.v. op til 0F. Dette har du allerede set i de forrige kapitler.

0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
1010 = A
1011 = B
1100 = C
1101 = D
1110 = E
1111 = F

Som du allerede har fået at se, så vil 4 bit blive sammensat til et ciffer i det hexadecimale talsystem. Tallet 10100101 binært, vil således bestå af to cifre hexadecimalt. Det ene er 1010 binært, mens det andet er 0101. Dette giver i alt det hexadecimale tal A5.

Nu tilbage til BCD-systemet. I dette system nøjes man med at bruge cifrene 0 - 9. Cifrene A til F er ulovlige cifre i BCD-systemet. Det største BCD-tal, som kan rummes i 8 bit bliver derfor 10011001 = 99 decimalt!

Lad os prøve at addere to BCD-tal, 33 og 44.

```
LD A,33h
ADD 44h
```

Vi bemærker, at selv om vi skriver 33h og 44h, så mener vi stadig regnet i BCD-systemet. Efter additionen vil A indeholde 77, og det er jo korrekt både med hensyn til lovlig notation i BCD-systemet og additionen. Hvis vi nu yderligere adderer 16

```
ADD 16h
```

vil register A ændre værdi til 8Dh (77h + 16h). Dette er et ukorrekt tal i BCD-notation. Hvad gør vi så? Vi bruger instruktionen DAA (Decimal Adjust Accumulator). Herefter vil register A indeholde 93h, og nu er BCD-notationen korrekt samtidig med, at resultatet af additionen decimalt er rigtigt.

Vi afprøver ovenstående i dette program:

Hex-kode	**Mnemonic**
3E33	LD A,33h
C644	ADD 44h
C616	ADD 16h
27	DAA
324B9C	LD (40011),A
C9	RET

Skriv i BASIC

```
CALL 40000 : PRINT PEEK (40011)
```

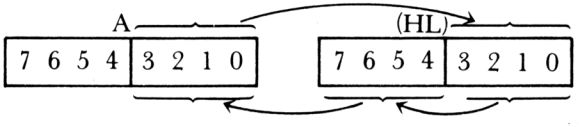
Resultatet er 147 = 10010011 i binær notation. Dette er omskrevet til BCD-aritmetik netop 93!

Når – eller hvis – du kommer til at arbejde med BCD-tal, skal du huske at bruge DAA efter enhver form for addition eller subtraktion for at sikre det korrekte resultat. Hvis de to BCD-tal fra starten er ulovlige, kan DAA selvfølgelig ikke stille noget op for at redde situationen.

I øvrigt benytter DAA N- og H-flagene til at finde frem til det rigtige resultat.

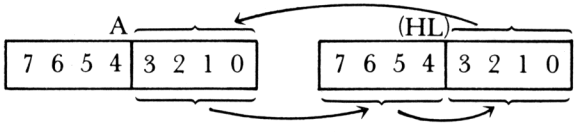
Som vi berørte før, så er der et par instruktioner til brug ved BCD-aritmetik. Disse hedder RLD (Rotate Left Digit) og RRD (Rotate Right Digit), og de flytter rundt på nibbles. En nibble be-

står af 4 bit, og der flyttes rundt mellem register A og indholdet på adresse HL, (HL). RLD-instruktionen fungerer således:



Som det ses på illustrationen herover, vil bit 0, 1, 2 og 3 i A flyttes til bit 0, 1, 2 og 3 i (HL). Dette sker ved, at bit 0 i A flyttes til bit 0 i (HL), bit 1 i A til bit 1 i (HL) o.s.v. Bittene 0, 1, 2 og 3 i (HL) flyttes til bit 4, 5, 6 og 7 i (HL), og disse bliver igen flyttet til bit 0, 1, 2 og 3 i A. Se tegningen.

Den anden hedder som anført RRD, og den udfører dette:



Bit 0-3 i A-registret flyttes til bit 4-7 i (HL). Disse bliver flyttet til bit 0-3 i (HL), og bittene dér er flyttet til bit 0-3 i A.

Hex-koderne for disse instruktioner er disse:

Hex-kode	**Mnemonic**
ED67	RRD
ED6F	RLD

Formålet med disse to instruktioner er, at man kan skifte en streng af BCD-tal i lageret. Se dette eksempel:

Hex-kode	**Mnemonic**
214D9C	LD HL,40013
3623	LD (HL),23h
3E56	LD A,56h
ED6F	RLD
324E9C	LD (40014),A
C9	RET

Når programmet køres, vil register A LOAD'es med 56 BCD-aritmetisk, og (HL) LOAD'es med 23. Det giver disse registre:

Register A: 5 6 (HL): 2 3

Nu udføres RLD, og A og (HL) ændrer udseende:

Register A: 5 2 (HL): 3 6

Efter rotationerne vil A således indeholde 52 BCD-aritmetisk, og (HL) indeholde 36. Da vi lader som om, BCD-systemet og dets tal er hexadecimalt, indeholder register A 52h = 82 decimalt og (HL) 36h = 54.

Prøv nu at eksekvere programmet med

```
CALL 40000 : PRINT PEEK (40013),PEEK (40014)
```

Du får formentlig udskriften 54 og 82 – netop hvad vi erfarede før, at A og (HL) skulle indeholde.

Lad os prøve at skifte en hel streng i lageret. Vi ser på dette eksempel:

```
På adresse 40024 :  0 0 =  0 i BCD-system
På adresse 40025 :  1 2 = 12 i BCD-system
På adresse 40026 :  3 4 = 34 i BCD-system
På adresse 40027 :  5 0 = 50 i BCD-system
```

Hvis vi prøver at lægge disse bytes ved siden af hinanden, får vi flg. billede:

```
0 0  1 2  3 4  5 0
```

Vi kan betragte dette som det seks-cifrede decimaltal 123.450. Vi ønsker nu at multiplicere dette tal med 10, altså rykke alle de fire bytes én halv gang (en halv byte – en nibble) til venstre, så det kommer til at se således ud:

```
0 1  2 3  4 5  0 0
```

Prøv følgende program:

Hex-kode	**Mnemonic**
210012	LD HL,1200h
22589C	LD (40024),HL
213450	LD HL,5034h
225A9C	LD (40026),HL
215B9C	LD HL,40027
AF	XOR A
0604	LD B,4
ED6F	RLD
2B	DEC HL
10FB	DJNZ – 5
C9	RET

Efter de 4 første LOAD-instruktioner, vil adresserne 40024 til 40027 se således ud:

Adresse	40024	40025	40026	40027
Indhold	0 0	1 2	3 4	5 0

40024 indeholder 00h, 40025 12h, 40026 34h, og endelig indeholder 40027 50h. Vi mener naturligvis stadig væk BCD-aritmetisk, og ikke hexadecimalt. Derefter LOAD'es HL med adressen til den bageste byte, hvorefter rotationsløkken starter. Inden første situation:

A = 0
(HL) = 50

Efter 1. rotation:

A = 05
(HL) = 00

hvorefter HL decrementeres, så

(HL) = 34

Efter 2. rotation:

A = 03
(HL) = 45

Fortsæt selv, så du kan få styr på rotationerne.

Indtast programmet på hex-loaderen og skriv derefter CALL 40000. Prøv så at få udskrevet indholdet på 40024-40027 i hexadecimal notation. Du vil få udskriften 01234500, og det er netop det resultat, vi ønskede (123.450 * 10).

Du har altså i BCD-aritmetik mulighed for at arbejde med tal større end de sædvanlige 65536 (2^{16}). Det virker måske en smule svært, og der er ingen tvang til at bruge BCD-systemet. Når du har fået lidt mere erfaring, kan du måske tage det op til overvejelse.

Vi vil afslutte dette kapitel med et program, som udskriver hex-koder. Det kan bl.a. være nyttigt, hvis man ønsker at se, hvad der står i RAM af maskinkoderutiner. Har man lavet en indtastningsfejl, vil dette program normalt afsløre den. Rutinen ser således ud:

Hex-kode	**Mnemonic**
21409C	LD HL,40000
0E2A	LD C,42
LØKKE 1	
7E	LD A,(HL)
57	LD D,A
07	RLCA
07	RLCA
07	RLCA
07	RLCA
0602	LD B,2
LØKKE 2	
E60F	AND 15
FE0A	CP 10
3802	JR C,2
C607	ADD 7
C630	ADD 48
CD5ABB	CALL TXT OUTPUT
7A	LD A,D
10F0	DJNZ LØKKE 2

```
3E20        LD A,32
CD5ABB      CALL TXT OUTPUT
CD5ABB      CALL TXT OUTPUT
23          INC HL
0D          DEC C
20DC        JR NZ, LØKKE 1
C9          RET
```

Efter CALL 40000 vil du få en række hexadecimale tal skrevet ud på skærmen – svarende til dem, du lige har indtastet! I rutinens begyndelse LOAD'er vi HL og C med hhv. startadressen for de bytes, der skal udskrives, og antallet af dem. Dernæst finder A ud af, hvilket tal der står på adresse HL. Så kommer selve programmet. Som nævnt under indledningen til BCD-aritmetik, er de 4 første bit i register A det første hexadecimale ciffer. Disse fire bit roteres nu hen til bit 0-3, mens bit 4-7 resættes med ordren AND 00001111. Register A indeholder altså nu et tal mellem 0 og 15, nemlig det første hexadecimale ciffer. Da tallene og bogstaverne ikke ligger lige ved siden af hinanden i karaktersættet, er vi nødt til at addere på den mærkelige måde, som er tilfældet. Tallene har koderne fra 48 til 57, mens bogstaverne A-F har koderne 65-70. Ved at sammenligne med 10 (CP 10) finder vi ud af, om værdien i A-registret hexadecimalt er et tal eller et bogstav. Hvis det er et tal, vil C-flaget blive sat. Hvis dette er tilfældet, vil instruktionen ADD 7 blive sprunget over i programudførelsen. Det første bogstav (A) har den decimale værdi 10, og den vil ved additionen med 48 derfor under normale omstændigheder give karakterkoden 58. Derfor tillægges yderligere 7 for at give karakterkoden 65, A. Derefter hoppes tilbage via DJNZ-instruktionen for at udskrive andet ciffer, og der foregår den samme proces. D-registret bruges til at gemme det rigtige tal, der skal udskrives, fra første ciffer til det andet ciffer. Efter DJNZ-instruktionen udskrives to mellemrum (CHR$ (32)), og LØKKE 1 kører om igen, hvis der er flere bytes, der skal udskrives.

Brug dette program til at lede efter indtastningsfejl.

Kapitel 12

Dette kapitel hører til i den sværere ende. De Z80-instruktioner, som vi her beskriver, er svære at bruge – og tilmed sjældent anvendt af brugerne. På Amstrad vil de være endnu sjældnere, da den har indbygget foranstaltninger, der praktisk talt eliminerer anvendelsen af disse instruktioner for en bruger. Vi vil alligevel gerne gennemgå dem for fuldstændighedens skyld.

Når en computer skal kommunikere med ydre enheder, såsom printer, skærm og båndoptager, så sker det via såkaldte porte. En port er således et sted, hvor kontakten til omverdenen skabes. Instruktionerne, der behandler portene, hedder IN og OUT.

Når indholdet på en port skal læses, skal adressen på denne først sendes ud på den såkaldte adressebus. Denne bus fylder 16 af Z80-processorens 40 ben, og det er derfor en 16 bits-adressebus. Alligevel vil en IN-instruktion kun aflæse fra en port specificeret som et 8 bits-tal.

Instruktioner, som aflæser indholdet på en port specificeret i C-registret ind i et vikårligt register, hedder:

Hex-kode	**Mnemonic**
ED78	IN A,(C)
ED40	IN B,(C)
ED48	IN C,(C)
ED50	IN D,(C)
ED60	IN H,(C)
ED68	IN L,(C)

Portens nummer ligger altså imellem 0 og 255, men adresseredes ikke en 16 bits-adressebus? Jo, og derfor skal man være opmærksom på, at indholdet i B-registret også lægges ud på adressebussen. B-registret vil således lægge sig på de 8 mest betydende bits, og C-registret på de mindst betydende.

Desuden findes instruktionen

Hex-kode	Mnemonic
ED70	IN F,(C)

Denne instruktion indlæser ikke indholdet på en port, men sætter kun flagene i henhold til portens indhold. Der berøres derfor ingen registre.

Alle de nævnte instruktioner påvirker flagene således:

S-flaget kopierer sign bit i værdien på porten
Z-flaget bliver sat, hvis portens indhold var nul
P/V-flaget virker som paritetsindikator
C-flaget påvirkes ikke.

Desuden findes en instruktion, som indlæser indholdet på en port n i A-registret (n er et tal mellem 0 og 255):

Hex-kode	Mnemonic
DBn	IN A,(n)

Denne instruktion påvirker ikke flagene. Naturligvis vil også denne adressere adressebussen som en 16 bits-bus, og derfor sendes indholdet i A-registret ud på bussens høje del, mens naturligvis n sendes ud på den lave.

Den til Amstrad medfølgende instruktionsbog påpeger, at denne instruktion ikke skal benyttes, men at man derimod skal bruge instruktionerne, der sender BC ud på adressebussen.

På tilsvarende måde som med IN-instruktionen kan man lægge en værdi i et register ud på den port, som C-registret peger på. Instruktionerne dertil hedder:

Hex-kode	Mnemonic
ED79	OUT (C),A
ED41	OUT (C),B
ED49	OUT (C),C
ED51	OUT (C),D
ED59	OUT (C),E
ED61	OUT (C),H
ED69	OUT (C),L

Desuden findes instruktionen

Hex-kode	**Mnemonic**
D3n	OUT (n),A

Instruktionerne for at lægge en værdi ud på en port påvirker overhovedet ikke flagene.

Som vi nævnte i indledningen, har Amstrad CPC-464 indbygget features, der medfører, at vi ikke skal bruge IN-instruktioner for at aflæse tastaturet (er der trykket på en tast?), udskrive på en printer etc. Disse indbyggede behageligheder vender vi tilbage til i kapitlerne efter dette.

Der findes endelig også instruktioner, der aflæser en port specificeret i C-registret, sender indholdet ind på den adresse, HL-registret peger på, incrementerer HL og decrementerer B. Ialt udføres altså denne lille rutine:

```
IN (HL),(C)
INC HL
DEC B
```

Bemærk, at ovenstående INC (HL),(C) ikke findes i Z80-instruktionssættet, men blot er medtaget af os for at tydeliggøre instruktionen. Den hedder INI (INput and Increment), og den udfører altså ovenstående program. Den påvirker også flagene. Imidlertid får S- og P/V-flaget uforudsigelige værdier, og C-flaget berøres ikke. Z-flaget vil være sat, hvis B bliver talt ned til nul, ellers resat.

Instruktionen INI kan naturligvis bringes til at repetere, indtil B bliver nul, og da hedder den INIR (INput, Increment and Repeat). Z-flaget vil derved altså være sat efter en INIR-instruktion, mens de øvrige flag påvirkes som ved INI-instruktionen.

På samme måde findes instruktionerne IND (INput and Decrement) og INDR (INput, Decrement and Repeat). Disse instruktioner virker som hhv. INI og INIR, men HL-registret decrementeres i stedet for at blive talt én op.

Koderne for disse 4 instruktioner er:

Hex-kode	Mnemonic
EDA2	INI
EDB2	INIR
EDAA	IND
EDBA	INDR

På tilsvarende måde findes 4 blok-instruktioner med OUT-instruktioner. Den første af dem, OUTI (OUT and Increment), udfører dette lille program:

```
OUT (C),(HL)
INC HL
DEC B
```

Indholdet på adresse HL sendes ud på porten C, mens HL derefter incrementeres, og B tælles én ned. Bemærk igen, at instruktionen OUT (C), (HL) ikke findes, men er medtaget for forståelsens skyld.

Skal der repeteres til B er blevet nul, hedder instruktionen OTIR (OuT, Increment and Repeat), og tilsvarende findes OUTD (OUT and Decrement) og OTDR (OuT, Decrement and Repeat). OUTD udfører dette program:

```
OUT (C),(HL)
DEC HL
DEC B
```

OTDR repeterer OUTD, til BC er blevet nul.

Bemærk også ved disse 8 blok-instruktioner, at hvis B-registret antager værdien 0 ved start, vil B efter en ikke-repeterende instruktion antage værdien 255. Udføres en repeterende blokinstruktion, vil den derfor blive udført 256 gange, da B decrementeres, inden der undersøges, om den er nul.

Hex-koderne for de 4 OUT-instruktioner er:

Hex-kode	Mnemonic
EDA3	OUTI
EDB3	OTIR

EDAB	OUTD
EDBB	OTDR

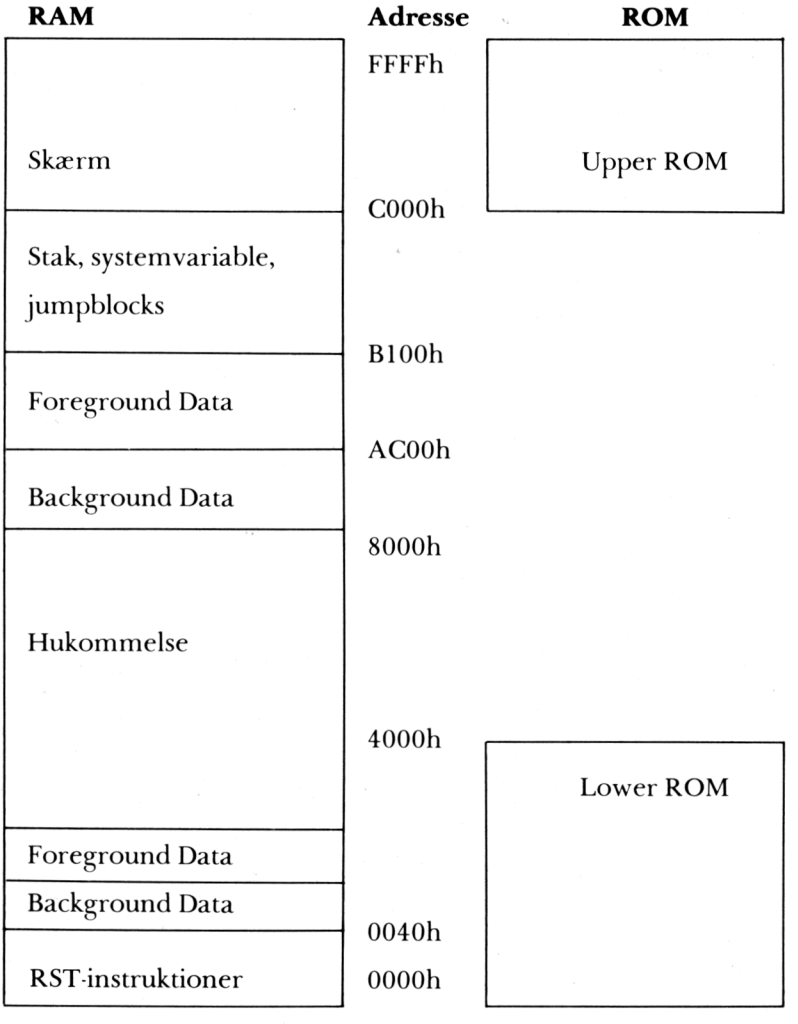
Flagenes påvirkning er analog til den, som vi gennemgik under IN-blok-instruktioner.

Du kan naturligvis selv eksperimentere med instruktionerne, og IN-instruktionerne er jo helt uskadelige, da portenes værdi blot aflæses. Du skal derimod være forsigtig med OUT-instruktionerne.

Kapitel 13

Z80A er en 8 bits-processor med en 16 bits-adressebus, og det betyder, at den kan adressere 65536 bytes. Derfor er det mærkeligt, at Amstrad CPC-464 indeholder både en 64 K RAM og 32 K ROM – for det giver 96 K, og det er altså ikke muligt adresseringsmæssigt. For at ændre dette har producenterne af Amstrad formået at lave et snedigt system, så man har adgang til alle 96 K – blot ikke alle på en gang. Dette system kaldes memory-mapping, og det går ud på, at man kan lægge RAM og ROM »ovenpå« hinanden. Ved hjælp af indbyggede rutiner kan man så bestemme, om man vil have fat i ROM eller i RAM.

Adresseringsområdet, d.v.s. det område indenfor hvilket Z80 kan operere, ser ud som på tegningen på næste side. Som det ses, er den 32 K store ROM opdelt i to 16 K ROM, nemlig »Upper ROM«, fremover kaldet UROM, og »Lower ROM«, kaldet LROM. Den øvre ROM kan blive lagt »ovenpå« RAM's øverste 16 K (adresse 49152-65535), mens den nedre ROM kan overlappe de nederste 16 K (0-16384). UROM indeholder BASIC-fortolkeren, der sikrer, at computeren kan kommunikere med os i BASIC, mens LROM indeholder operativsystemet, ofte kaldet »firmware«. Det er denne del, der tager sig af tastaturet, sørger for at sende data ud på skærmen, genererer lyde samt kommunikerer med ydre enheder såsom kasettebåndoptager og printer.



Amstrad kan som allerede nævnt ikke arbejde med både RAM og ROM på en gang, og derfor har den en mekanisme indbygget, der gør, at den kun har med den ene at gøre ad gangen. Enten er det RAM eller også er det ROM.

Når man i BASIC bruger PEEK på en adresse, hvor RAM og ROM overlapper hinanden, så er det RAM, man PEEK'er. Man kan således i BASIC ikke finde ud af, hvad der står i ROM.

Imidlertid indkobler og udkobler Amstrad selv ROM fra adresseringsområdet og maskinkodeprogrammøren har også mulighed for det. Der findes rutiner i de såkaldte jumpblocks, som vi omtaler om lidt, der opererer med dette træk. Vi kan således lave dette program:

Hex-kode	**Mnemonic**
CD06B9	CALL ENABLE LROM
110040	LD DE,16384
210000	LD HL,0
010040	LD BC,16384
EDB0	LDIR
C9	RET

Vi starter med at indkoble LROM på adresseringsområdet, hvorefter de 16 Kbytes er tilgængelige. Derefter udfører vi en LDIR-ordre, hvor vi overfører hele LROM fra adresserne 0-16383 til adresserne 16384-32767. Dernæst returnerer vi, og vi kan nu PEEK'e den overflyttede LROM. Vi kan gøre det med dette program – ændr startadressen for din hex-loader (til f.eks. 41000):

Hex-kode	**Mnemonic**
210040	LD HL,16384
7E	LD A,(HL)
FE20	CP 32
D45ABB	CALL NC,TXT OUTPUT
23	INC HL
CB7C	BIT 7,H
C0	RET NZ
18F4	JR – 12

Kører du dette program efter det andet, vil du få udskrevet LROM som karakterer, og her vil du faktisk på et tidspunkt kunne iagttage, at der står de ord og bogstaver, som Amstrad udskriver ved reset.

Du kan evt. selv prøve at lave et program, der udskriver LROM som tal. Derved vil du finde ud af, at den består af temmelig mange JP-instruktioner, der får computeren til at springe til andre steder i ROM.

RAM's opbygning er lidt mere kompliceret. På de nederste 64 bytes ligger en kopi af ROM's nederste 64 bytes, og vi skal om lidt se nærmere på, hvad disse indeholder af gode sager. Området fra 0040h til B100h indeholder først og fremmest det område, hvori programmer og variabler lægges. På vor tegning er desuden inkluderet 4 områder, kaldet »Foreground Data« og »Background Data«. Disse skal vi ikke bekymre os om, men blot være klar over, at de er der.

De øverste 16 K indeholder oplysninger om, hvordan skærmbilledet er bygget op. Vi skal i kapitel 15 gå ind på, hvordan de enkelte punkter kan findes og derved forklare, hvordan de 16384 bytes er fordelt.

Området fra B100h til C000h indeholder bl.a. maskinstakken, d.v.s. den stak, hvorpå vi PUSH'er og POP'er, og den ligger som allerede beskrevet nedad i RAM. Den starter omkring adresse 49152. Hvis du selv opretter nye stakke i et program, skal du ikke lægge dem hverken i de nederste eller øverste 16 K, da der da kan blive problemer med de indkoblede ROM.

Området indeholder dog også – hvad mere vigtigt er – 4 jumpblocks. I disse ligger rutiner, som maskinkodeprogrammøren med fordel kan kalde.

Den første af de 4 blokke, kaldet »the main jumpblock«, ligger fra adresse BB00h til BD39h. Den indeholder rutiner til behandling af tastaturet, tekstskærmen, grafikskærmen, kassettebåndoptageren, lydprocessoren m.m. Det er specielt denne blok, som vi vil behandle dybtgående i de kommende kapitler. Alle rutiner i blokken er på tre bytes, nemlig indeholdende JP-instruktioner til operativsystemet. Desuden er det værd at bemærke, at UROM er frakoblet, da behandling af skærm beliggende i de øverste 16 K er nødvendig. LROM er naturligvis tilkoblet.

Den anden blok, kaldet »Firmware Indirections«, ligger fra BBCDh til BDF3h, og den indeholder bl.a. rutiner til tekst- og grafik.

Den tredje blok hedder »The High Kernel Jumpblock«, og heri er rutiner til til- og frakobling af ROM indenfor adresseringsområdet.

Den fjerde og sidste blok er »The Low Kernel Jumpblock«, og den går fra adresse 0 til 63. Denne vil vi beskrive i det følgende.

Vi har i Z80-instruktionssættet 8 instruktioner, som vi ikke har

omtalt endnu. De hedder RST-instruktionerne (ReSTart-instruktionerne). De svarer til en CALL-instruktion, men adressen de kalder er fastlagt i forvejen. Udføres en RST 0, vil adresse 0 kaldes, udføres RST 08h, vil adresse 8 blive kaldt o.s.v. Disse 8 rutiner kan kaldes med én-byte-instruktioner, så ofte benyttede rutiner kan kaldes hurtigt. Det er derfor ikke helt tilfældigt, at der i de nederste 64 bytes ligger nyttige rutiner.

De 8 RST-instruktioner har flg. hex-koder:

Hex-kode	**Mnemonic**
C7	RST 00h
CF	RST 08h
D7	RST 10h
DF	RST 18h
E7	RST 20h
EF	RST 28h
F7	RST 30h
FF	RST 38h

Vi vil nu tage dem én efter én og beskrive deres funktion, og hvordan vi kan drage nytte af dem.

RST 0h svarer altså til en CALL 0, og det er ensbetydende med et totalt reset af systemet. Når computeren tændes, vil den automatisk starte på adresse 0. RST 0h kan vi således ikke drage særlig nytte af.

RST 08h bruges til at springe til et sted i området 0-16383 enten i RAM eller i ROM. De to bytes efter CFh i hex-koden (CF er hex-koden for RST 08h) skal indeholde adressen på den rutine, man ønsker at springe til. Grunden til, at man kun kan springe til rutiner indenfor de første 16384 bytes, er, at bit 15 og 14 ikke hører med til adressen. Disse angiver noget helt andet. Hvis bit 15 er sat, vil UROM være udkoblet (resat bit betyder tilkoblet), og hvis bit 14 er sat, vil LROM være udkoblet (resat medfører naturligvis tilkobling).

Inden du kalder RST 08h, skal du sørge for, at der øverst på stakken ligger den adresse, som rutinen, du lige har kaldt indirekte, skal vende tilbage til. Eksempel:

Hex-kode	**Mnemonic**
21479C	LD HL,40007
E5	PUSH HL
CF	RST 08h
FF7F	DEFB
3EFF	LD A,255
CD5ABB	CALL TXT OUTPUT
C9	RET

Lav linie 20 i hex-loaderen om til:

```
20 ADR = 16383 : MEMORY 16382
```

Indtast så:

Hex-kode	**Mnemonic**
3EFE	LD A,254
CD5ABB	CALL TXT OUTPUT
C9	RET

Vi starter i første rutine med at lægge adressen 40007 op på stakken. Dernæst udfører vi en RST 08h. De to bytes bagefter er FF7Fh – det er 01111111 11111111 i binær notation. Det betyder, at vi kobler LROM ud, kobler UROM ind (da vi skal afslutte med at returnere til BASIC), og så kalder adresse 16383, hvor vi har vores lille ekstrarutine liggende klar. Her udskrives CHR$ (255), hvorefter der hentes en returadresse på stakken. Bemærk, at RST-instruktionen ved sin udførelse ikke selv lægger nogen returadresse på stakken. Vore 40007 på stakken lægges altså ned i PC, og herfra fortsættes programmet: CHR$ (254) udskrives, og der returneres til BASIC.

Vi prøver med CALL 40000 – og der kommer CHR$ (255) og CHR$ (254) oppe på skærmen. Nydeligt . . .

Prøv så med CALL 40004. Ups . . . Der kommer kun CHR$ (254). Vi har jo ingen returadresse på stakken, som vi kan hente ned, så efter udskrivningen af CHR$ (254) returneres blot til ROM (det er jo denne returadresse, der står øverst på stakken).

Udover de rutiner, som kan kaldes med en RST-instruktion, er der i de nederste 64 bytes også placeret andre rutiner, der er

nært beslægtet med RST-rutinerne. Disse vil vi også behandle i dette kapitel, så redegørelsen ikke blot bliver sporadisk. Den første af disse »skæve« rutiner ligger på adresse 0Bh, og kan således kaldes med CALL 11.

Denne rutine svarer faktisk til en RST 08h, men den skal ikke efterfølges af to bytes, der viser, hvor der skal udføres en rutine. I stedet står denne rutines startadresse i HL-registret, når CALL 11 udføres. Du skal ikke PUSH'e nogen returadresse op på stakken denne gang, inden CALL 11 udføres, da denne jo i sig selv lægger en returadresse derop.

For både rutinen på adresse 8 og 11 gælder, at alle registre og flag kører uberørt videre til den rutine, som egentlig kaldes.

På adresse 14 starter en ny rutine. Vi har tidligere set denne rutine – nemlig i kapitlet om CALL-instruktionen. I Z80-instruktionssættet findes som bekendt instruktionen JP (HL). På adresserne 14 og 15 står to instruktioner, nemlig

```
PUSH BC
RET
```

Hvis vi derfor bruger CALL 14, så udføres egentlig en CALL (BC) – en helt ny mulighed! og udføres en JP 14, så er der tale om JP (BC), som vi allerede har set det.

RST 10h er til brug, når man har flere ROM. Da vil den kalde en af disse (maksimum 4), og adressen, den kalder, er udtrykt i de to bytes efter RST 10h-instruktionen. De nederste 14 bit indeholder startadressen, der automatisk tillægges C000h (49152), før der kaldes. De to mest betydende bit (bit 14 og 15) indeholder nummeret på den af de 4 ROM (2 bit kan netop udtrykke 4 værdier, 0-3), som man vælger at arbejde med. Denne kobles herved automatisk ind, og desuden kobles LROM ud. Når der returneres fra den rutine, som kaldes, genetableres ROM's status fra før vi udførte RST 10h.

Rutinen på adresse 19 svarer til RST 10h, men i stedet for at benytte to bytes efter selve instruktionen, skal adressen og nummeret på den valgte ROM være placeret i HL, når CALL 19 udføres.

For begge rutiner gælder, at det kun er IY-registret, som berøres (ændrer værdi).

På adresse 22 står en rutine analog med den på adresse 14. Den arbejder dog i stedet for med DE-registret. De to bytes (på adresserne 23 og 24) har værdierne 213 og 201, ensbetydende med

```
PUSH DE
RET
```

Man kan således også med denne rutine udføre CALL (DE) og JP (DE). Registrenes og flagenes værdi berøres naturligvis ikke – PUSH DE og RET berører jo ingen af disse ting.

RST 18h bruges til at kalde et eller andet sted i ROM eller RAM. Instruktionen skal efterfølges af 3 bytes, hvoraf de to første angiver en adresse mellem 0 og 65535, mens den sidste angiver hvilken status, UROM og LROM er i. Denne sidste adresse kan således antage værdier mellem 0 og 255, og de udtrykker følgende:

0-251: vælg ROM – tilkobling af UROM, frakobling af LROM
252: tilkobling af UROM – tilkobling af LROM
253: tilkobling af UROM – frakobling af LROM
254: frakobling af UROM – tilkobling af LROM
255: frakobling af UROM – frakobling af LROM

Der kan i alt være 252 ROM, og med en angivelse mellem 0 og 251 vælges én af disse. Ønskes ingen ændring i valget af ROM, benyttes én af de fire sidste værdier.

Rutinen på adresse 27 svarer til en RST 18h, men der benyttes ingen parameter. I stedet er adressen, der skal kaldes, i HL-registret, og ROM-status i C-registret.

Rutinen på adresse 30 fylder kun én byte:

```
JP (HL)
```

Derfor vil JP 30 være det rene spild (den fylder tre bytes, JP (HL) i sig selv jo kun én), hvis man vil springe til adressen i HL. Derimod kan CALL (HL) udføres med CALL 30.

RST 20h udfører ganske simpelt en LD A,(HL)-instruktion, blot kigges altid i RAM, hvadenten ROM er koblet til eller koblet fra.

Det betyder, at selv med UROM koblet til i nedenstående eksempel vil man kigge i RAM.

Hex-kode	**Mnemonic**
2101C0	LD HL,49153
E7	RST 20h

Denne programstump vil således medføre, at A-registret indeholder værdien på plads i 49153 i *RAM*!

Når man kalder rutinen på adresse 35, skal HL indeholde adressen på en 3 bytes-blok. I denne 3 bytes-blok skal først stå adressen på den rutine, som skal kaldes (det fylder 2 bytes), og den sidste skal indeholde ROM-status som beskrevet i forbindelse med rutinen på adresse 24. Rutinen på adresse 35 ødelægger IY-registret.

RST 28h bruges til at springe til LROM eller de midterste, centrale 32 K RAM (dem fra adresse 16384 til 49151). Instruktionen skal efterfølges af to bytes, der angiver, til hvilken adresse der skal springes. LROM er altid koblet til, inden den egentlige rutine kaldes, men den kobles fra, når rutinen returnerer. På stakken skal stå en værdi, som angiver, hvortil rutinen skal returnere, når den er færdig.

RST 30h er en rutine, der står helt til dit eget brug. De 8 bytes fra adresse 48 til 55 kan du selv disponere over og indskrive dine egne instruktioner i. LROM bliver koblet fra, når RST 30h kaldes. Vi tager et eksempel: hvis du har brug for tit at få en bestemt karakter ud, så kan du lægge rutinen herunder fra adresse 48:

Hex-kode	**Mnemonic**
3EFC	LD A,252
CD5ABB	CALL TXT OUTPUT
C9	RET

Rutinen, der kalder RST 30h, kunne være (taget fra adresse 40000 som sædvanlig):

Hex-kode	**Mnemonic**
06FF	LD B,255
F7	RST 30h

```
10FD        DJNZ - 3
C9          RET
```

CALL 40000 vil da medføre, at du får udskrevet din karakter (her CHR $(252)) 255 gange.

Man kan selvfølgelig også lægge en JP-ordre derned, så man derved kan få kaldt længere rutiner.

RST 38h og rutinen på adresse 59 kort derefter vil vi ikke gøre rede for. De behandles senere i vort kapitel om interrupts. Det er nemlig således, at når der kommer et interrupt (en afbrydelse), vil der blive udført en RST 38h-instruktion automatisk – dog kun i interrupt mode 1. Det vender vi tilbage til.

Kapitel 14

Vi skal nu til at se lidt nærmere på tastaturet. Til dette formål er der i den omtalte jumpblock 26 rutiner til behandling af tastaturet.

Du har sikkert bemærket, at Amstrad kan huske, hvilken tast du har trykket på, og først udskriver den tilhørende karakter, når den får tid. Det kan f.eks. være sådan, at du skriver hurtigere, end Amstrad kan udskrive på skærmen. Alligevel får du altså udskrevet alt, hvad du har indtastet – til en vis grænse!

Det fænomén, at Amstrad kan huske nedtrykninger, skyldes, at når computeren aflæser tastaturet, bliver en information om hvilken tast der er nedtrykket lagt i en kø – en såkaldt keybuffer. Når Amstrad udskriver en karakter, oversættes informationen forrest i køen til et tastenummer og den dertil hørende karakter udskrives. I bufferen står således information om, hvilke taster, du har trykket på, men som du endnu ikke har set nogen reaktion på (på skærmen).

Den første rutine, vi vil vise dig, er den såkaldte KM WAIT CHAR. Denne returnerer den første karakter i keybufferen i A-registret, og C-flaget sættes. Hvis der ikke er nogen karakter, returnerer rutinen ikke, men afventer, at der kommer én (i keybufferen). Vi kan teste den med dette lille program:

Hex-kode	**Mnemonic**
CD06BB	CALL KM WAIT CHAR
FE0D	CP 13
C8	RET Z
CD5DBB	CALL TXT WR CHAR
18F5	JR – 11

Rutinen KM WAIT CHAR ligger på adresse BB06h i den første jumpblock. Vi kalder denne rutine i starten, og når den returnerer, ligger den første karakter i keybufferen i A-registret. Vi sam-

menligner den med 13 – karakterkoden for ENTER – og er de lig hinanden, sættes Z-flaget, og der returneres. Ellers udskrives karakteren med rutinen TXT WR CHAR, der minder meget om TXT OUTPUT. Forskellen skal vi komme ind på i næste kapitel. Når karakteren, som den pågældende tast forårsager, er udskrevet, starter hele rutinen forfra igen.

Vi beder dig om at bruge din medfølgende instruktionsbog til at finde numrene på tasterne, karakterkoderne for tasterne i hhv. normal, skiftet (SHIFT'ed) og CONTROL-tilstand, samt koderne for de såkaldte »expansion characters«, udvidelseskaraktererne. Disse karakterer er de 32, som man selv kan definere til, ved et tryk på en tast, at udskrive flere karakterer. Disse ligger i området fra 128 til 159.

I vort programeksempel fra før vil udskrivning af en sådan »udvidelseskarakter« ikke volde vanskeligheder. KM WAIT CHAR vil blive kaldt, og alle karaktererne for denne ene tast, som man selv har defineret, vil blive lagt i keybufferen. Så udskrives den første, og rutinen startes om. Nu venter rutinen dog ikke på det næste tryk på tastaturet, for i keybufferen står jo flere karakterer! Den udskriver derfor frejdigt endnu én af disse. Sådan bliver den naturligvis ved, til hele udvidelseskarakteren er udskrevet.

Vi kan illustrere det med et eksempel. Fra start – ved reset af computeren – vil CONTROL + den lille ENTER (kode 140) give dette:

```
RUN"
Press PLAY then any key:
```

Der sker altså det samme, som hvis du skriver

```
RUN" (ENTER)
```

Vi kan nu prøve dette program:

Hex-kode	**Mnemonic**
CD06BB	CALL KM WAIT CHAR
CD5DBB	CALL TXT WR CHAR
C9	RET

Kør dette program, og tryk så på CONTROL + den lille ENTER. Det giver udskriften R, nemlig den første karakter i keybufferen ved udskriften. Imidlertid er bufferen jo ikke tømt, hvorfor der i BASIC udskrives

```
UN" (ENTER)
```

hvilket medfører syntaks-fejl . . .

Du skal desuden være klar over, at når man kalder en rutine i en jumpblock, så kan man ikke altid være sikker på, at registrene beholder deres værdier. Det kan jo være, at rutinen nede i LROM selv bruger registrene. Vi vil under gennemgangen af hver enkelt rutine – når vi finder det formålstjenligt – skrive hvilke registre og flag, der bevares, og hvilke, der ødelægges. Disse oplysninger står i øvrigt naturligvis også i kapitel 21.

I KM WAIT CHAR får A-registret værdien af karakteren, alle andre registre berøres ikke, C-flaget sættes, og resten af F-registret beholder ikke sine værdier.

Lige ved siden af KM WAIT CHAR, på adresse BB09h, ligger KM READ CHAR, og den er nært beslægtet med netop KM WAIT CHAR. I modsætning til denne venter KM READ CHAR ikke på, at der dukker en karakter op i keybufferen. Den returnerer øjeblikkelig – enten med den første karakter i keybufferen eller uden karakter (hvis der ikke var nogen). For os kan forskellen iagttages på C-flaget. Hvis dette er sat, er der fundet en karakter i keybufferen, og A-registret indeholder dens kode. Er C-flaget derimod resat, så var der ingen karakter i bufferen, og A-registrets indhold bliver ubrugeligt (indholdet bliver 0). Alle de andre registre beholder også her deres værdier, mens de andre flag berøres. Afprøv selv denne rutine, men husk at tage højde for, om C-flaget er sat eller resat. Der skal kun udskrives karakterer, hvis det er sat.

Den næste rutine hedder KM SET EXPAND, og den omhandler netop udvidelseskaraktererne, som vi allerede har beskrevet. Denne rutine definerer en udvidelseskarakter. B indeholder den kode, som den har (140 vil altså være koden for CONTROL + lille ENTER), C indeholder længden af udvidelseskarakteren, mens HL peger på den adresse, hvor første tegn i strengen ligger. Vi afprøver det i dette lille, men yderst effektive eksempel:

Hex-kode	Mnemonic
214A9C	LD HL,40010
011180	LD BC,32785
CD0FBB	CALL KM SET EXPAND
C9	RET
424F5244	DEFB
45522030	DEFB
3A494E4B	DEFB
20302C360D	DEFB

Vi bruger karakterkoden 128 – svarende til 0 på det numeriske tastatur – og vores streng er placeret på de 17 bytes liggende fra 40010 til 40026. Disse tre oplysninger er lagt i BC- og HL-registrene, inden rutinen KM SET EXPAND på BB0Fh kaldes. Prøv nu programmet med CALL 40000. Tryk så på 0 på det numeriske keyboard. Voilà . . .

Vores 17 karakter lange streng ser således ud:

BORDER 0 : INK 0,6 (ENTER)

Bemærk den sidste karakter, nemlig den med koden 13. Det er en kontrolkode; den for ENTER. Denne er skyld i, at vi – når vi bruger 0 – ikke blot får skrevet strengen ud på skærmen, men sandelig også eksekveret den.

Prøv selv med andre strenge og andre taster.

Rutinen har faktisk en ekstra lille fordel indbygget, idet den tester, om operationen er lykkedes. C-flaget sat meddeler, at alt er i orden, mens resat tilstand betyder, at der forekom ulovlige ting. Desuden ændres indholdet af både A, BC, DE og HL, mens de andre registre beholder deres værdier. Flagregistret berøres ligeledes.

Den næste rutine, som vi vil omtale, hedder KM TEST KEY, og den er beliggende på adresse BB1Eh. Med den kan man teste, om en tast er nedtrykket eller ej. Man giver A tastenummeret (ikke karakterkoden, men nummeret på tasten), og rutinen returnerer med enten Z-flaget sat eller resat. Er det resat, vil tasten være nedtrykket i det øjeblik, hvor rutinen aflæser tastaturet, mens det vil være sat, hvis den ikke er. Eksempel:

Hex-kode	Mnemonic
21549C	LD HL,40020
0603	LD B,3
E5	PUSH HL
7E	LD A,(HL)
CD1EBB	CALL KM TEST KEY
E1	POP HL
28F8	JR Z, – 8
23	INC HL
10F5	DJNZ – 11
CD03BB	CALL KM RESET
C9	RET
3E1B3E	DEFB

Indtast rutinen på adresse 40000, som sædvanlig, og med CALL 40000 starter du så programmet.

Først LOAD'es HL-registret med 40020. Det er starten på vores tre små bytes i slutningen, der indeholder data. Dernæst får B værdien 3 svarende til antallet af data. Senere dukker en DJNZ op, så vi bruger altså B-registret som tæller. Løkken skal således gennemløbes tre gange. Først PUSH'es HL, da rutinen KM TEST KEY berører dette register. Dernæst LOAD'es A med indholdet på adressen i HL, d.v.s. ved start får A værdien 62 (værdien på adresse 40020), der er et tastenummer. I din instruktionsbog kan du se, at det er nummeret på tasten C. Så kaldes KM TEST KEY, og den returnerer med Z-flaget som udslagsgivende faktor, for hvorvidt der er blevet trykket på C-tasten. Så POP'es HL tilbage, og vi går tilbage til PUSH HL og henter den *samme* værdi (første gang 62), hvis ikke C er nedtrykket. Er C derimod det, vil Z-flaget være resat. Da incrementeres HL til næste byte i data-blokken, hvorefter der hoppes tilbage, hvis ikke alle data er brugt.

Bemærk i øvrigt, at vi bruger endnu en rutine i den første jumpblock, nemlig KM RESET. Denne tømmer bl.a. keybufferen. For at se dens fuldstændige formål må du slå op i kapitel 21. Hvorfor nu tømme keybufferen? Jo, for ellers vil de karakterer, som fremkommer ved alle de tastetryk, som vi har lavet undervejs i rutinens forløb, blive udskrevet, når der returneres. Prøv at fjerne instruktionen CALL KM RESET.

Dette program kræver altså, at man indtaster et kodeord for at

komme ud af det og tilbage i BASIC. Find selv kodeordet ved at gennemgå programmet og forstå vor forklaring ovenover.

Programmet kan naturligvis integreres i større programmer og derved bruges som stopklods, hvis man ikke ønsker, at andre personer skal »rode« i et program.

Vi nævnte, at HL berøres af KM TEST KEY, men også A og flagene ændrer indholdet. De øvrige registre er bevaret – i rutinen PUSH'er vi således heller ikke værdien af B!

I jumpblocken findes også en rutine, der behandler evt. tilsluttede joysticks. Der kan tilsluttes to joysticks til Amstrad, og rutinen returnerer med status for dem begge.

Rutinen hedder KM GET JOYSTICK, og den ligger på BB24h. Der er ingen restriktioner, som skal være overholdt, når rutinen kaldes. Når den returnerer, vil A og H indeholde status af 1. joystick (værdierne i A og H er helt ens), mens L indeholder status af det 2. joystick. I registrene har de enkelte bits betydning:

Bit 0 sat = sat
Bit 1 sat = ned
Bit 2 sat = venstre
Bit 3 sat = højre
Bit 4 sat = knap 2
Bit 5 sat = knap 1
Bit 6 = bruges ikke
Bit 7 = altid nul

Joystickrutinen kan specielt benyttes til spil. Bemærk i øvrigt, at det 1. joystick har 6 forskellige tastenumre, alt efter om det er oppe, nede o.s.v. Det andet joystick har derimod tastenumre tilfælles med nogle af de andre taster på tastaturet. Det kan ikke skelnes fra disse. Tastenumrene kan du se i din instruktionsbog.

At de har forskellige tastenumre, alt efter hvordan styrepinden står, betyder, at du kan bruge rutinen KM TEST KEY for at undersøge om pinden eksempelvis er oppe.

Som vi allerede har gjort opmærksom på, så kan du finde mange flere rutiner i kapitel 21. Ialt findes der 26 rutiner til behandling af tastaturet, og vi har kun i dette kapitel omtalt de af dem, som vi har fundet vigtigst. Udover de her beskrevne kan nævnes, at man kan sætte repetitionshastighed, ventetiden før repetitio-

nen, omdefinere taster (tildele dem andre karakterer) i både normal-, SHIFT- og CONTROL-tilstand og meget mere. Prøv naturligvis selv at eksperimentere videre, og brug så kapitel 21 som opslagsværk.

Kapitel 15

Vi vil i dette kapitel beskæftige os med tekstskærmen, og mulighederne med denne er så mangfoldige, at det vil føre for vidt at omtale det hele her. Vi vil forklare dig de mest nyttige rutiner og give små eksempler på dem.

Skærmen ligger, som allerede nævnt og forklaret i kapitel 13, fra adresse 49152 til 65535. Heri er der ialt 16384 bytes, og skærmen benytter de 16000 af dem. De sidste 384 er tilovers.

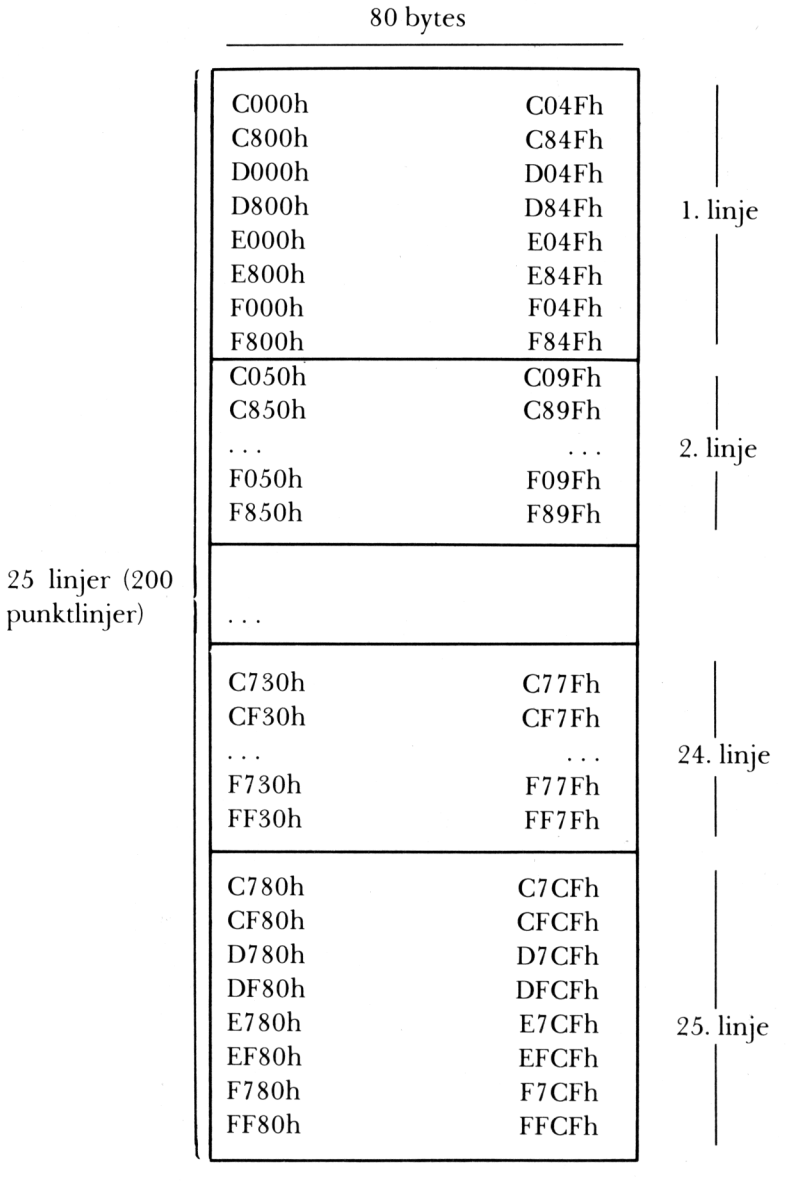
Når man arbejder i mode 2, så er der 80 karakterer på en linje, og med 25 linjer giver det i alt mulighed for 2000 tegn på skærmen. Imidlertid fylder et 1 tegn 8 bytes. Det er bygget af en 8x8 matrix. Som du kan iagttage det i din instruktionsbogs appendiks III, vil bogstavet A se således ud:

```
   11
  1111
11   11
11   11
111111
11   11
11   11
```

Med 0 indsat på de tomme pladser fås:

```
00011000
00111100
01100110
01100110
01111110
01100110
01100110
00000000
```

SKÆRMENS OPBYGNING:



Den første vandrette linje i A'et fylder således én byte i mode 2, og den har værdien 24. Således også med de andre linjer, så A'et i alt fylder de nævnte 8 bytes – én byte for hver linje.

Med et tegn fyldende 8 bytes og 2000 tegn på skærmen fås, at skærmen bruger 16000 bytes, som vi allerede har omtalt. Tegningen på forrige side viser, hvorledes skærmen er opbygget.

Hele første vandrette linje ligger fra C000h til C04Fh. Dernæst ligger den niende vandrette linje på skærmen (nemlig 2. rækkes første vandrette linje) fra C050h til C09Fh. Således fortsættes over alle skærmens 25 rækker, til de første 2000 bytes er brugt. Så ligger der et område på 48 bytes – første gang fra C7D0h til C7FF – som ikke bruges. Efter dette, på C800h, begyndes så på 2. vandrette linje o.s.v. Skærmen består altså således af 2000 bytes fra de første vandrette linjer i alle karakterer + 48 ubrugte bytes + de 2000 bytes fra næste vandrette linjer i alle skærmens karakterer o.s.v., indtil der er brugt 8 * (2000 + 48) = 16384 bytes.

Vi kan demonstrere skærmens opbygning med dette BASIC-program:

```
10 MODE 1
20 FOR N = 49152 TO 65535
30 POKE N,240
40 NEXT N
```

Kør programmet, og du vil sikkert forstå vore forklaringer uden problemer. Bemærk endvidere, at vi også POKE'er de 384 bytes, som ikke benyttes. Prøv i øvrigt at bytte de 240 i linje 30 ud med 255, og kør igen! Dermed vil du få en rød skærm. Du kan således POKE'e ialt 256 forskellige farvekoder ind. Opbygningen af disse er meget indviklet.

Vi har i programmet indsat en MODE 1-kommando for, at adresse 49152 vil svare til skærmens øverste venstre hjørne. Det kan man nemlig ikke altid regne med. Når skærmen scrolles – ruller op eller ned – vil dette forhold ændre sig, og det skal man være på vagt overfor, hvis man POKE'er direkte ind i skærmbilledets hukommelse. Du kan prøve BASIC-programmet uden linje 10, men husk så at scrolle inden du eksekverer programmet. Efter MODE-kommandoen vil adresse 49152 altid være skærmens øverste venstre hjørne.

Imidlertid vil en maskinkodeprogrammør ikke spises af med et BASIC-program. Hvorfor ikke gå til den ægte vare:

Hex-kode	**Mnemonic**
2100C0	LD HL,49152
36F0	LD (HL),240
23	INC HL
CB7C	BIT 7,H
C8	RET Z
18F8	JR – 8

Prøv det – og du ser en brøkdel af maskinkodens effektivitet! Det går mange gange hurtigere end i BASIC. Programmet i sig selv må være selvforklarende – dog kan vi lige sige, at bit 7 i H bliver resat, når H når værdien 0. Da er vi færdige, og der returneres.

Vi så før, at koden for A's første vandrette linje var 24 i mode 2. Som du ved, kan man i mode 2 kun bruge to farver, og det skyldes, at en karakter kun skal fylde én byte, for ellers kan skærmen ikke være på den afsatte plads. Komplementerer man farverne, vil A's første vandrette linje imidlertid få værdien 231 (255–24).

Bruges mode 1, så er der kun 40 karakterer på en række. Dette betyder, at én karakter fylder to bytes, og den kan derved udtrykke 4 farver efter et bestemt bitmønster, som den selv kan dechifrere. I mode 0 med kun 20 karakterer pr. række vil en karakter fylde 4 bytes, og det giver mulighed for at arbejde med 16 farver. Alt dette ordner Amstrad internt, uden at vi behøver blande os i det.

Den første rutine, som vi vil omtale, hedder SCR SET MODE, og den er fra den del af den største jumpblock, der omhandler skærmen. Med denne rutine kan man skifte tilstand mellem mode 0, 1 og 2. Med kald af denne rutine følger, at skærmen renses, alle vinduer fjernes, grafiknulpunktet (origo) sættes i nederste, venstre hjørne og meget mere. Som vi allerede har sagt, vil kaldet af denne rutine også medføre, at adresse 49152 vil indeholde skærmens øverste venstre hjørne.

Rutinen ligger på BC0Eh, og A-registret skal indeholde enten 0, 1 eller 2 for den ønskede tilstand. Ellers sker der intet. Prøv selv rutinen.

Der findes to rutiner til at udskrive karakterer med, og du har

allerede stiftet bekendtskab med dem begge. De hedder TXT OUTPUT og TXT WR CHAR og er beliggende på hhv. BB5Ah og BB5Dh. Karakteren i A-registret udskrives på den nuværende printposition til den nuværende stream. Der er dog en væsentlig forskel på rutinerne, og det demonstreres med dette program:

Hex-kode	**Mnemonic**
3E07	LD A,7
CD5ABB	CALL TXT OUTPUT
CD5DBB	CALL TXT WR CHAR
C9	RET

Når du kører programmet, så husk at skrue op for den indbyggede højttaler. Du vil høre en lyd og et rumvæsen vil dukke op på skærmen. Som vi før har beskæftiget os med, er der kontrolkoder liggende fra CHR $ (0) til CHR $ (31), og de har alle deres bestemte formål, som du kan iagttage i appendiks E. Forskellen på de to rutiner er, at TXT OUTPUT opfatter værdien mellem 0 og 31 i A-registret som kontrolkoder, mens TXT WR CHAR udskriver karakteren. Endvidere er det værd at bemærke, at TXT OUTPUT ikke berører nogen registre, men returnerer med de samme værdier, som da rutinen blev kaldt. TXT WR CHAR derimod ødelægger både A, BC, DE, HL og flag-registret.

Den næste rutine hedder TXT SET CURSOR, og med den kan man få placeret sin markør på en hvilken som helst plads på skærmen. H skal indeholde kolonnen, og L skal indeholde linjen, hvorpå markøren skal placeres. Arbejder man med streams – vinduer – er det værd at bemærke, at koordinaterne i HL-registret er sat i forhold til vinduet i den nuværende streams øverste venstre hjørne.

Rutinen ligger på adresse BB75h, og den destruerer såvel A og HL som F-registret.

På adresse BB60h ligger en rutine kaldet TXT RD CHAR, og med den kan vi finde ud af, hvilken karakter der står på den nuværende markørposition i den nuværende stream. Der kræves ingenting, før man kalder rutinen. Til gengæld returnerer den med et indhold i A-registret og C-flaget som udslaggivende faktor for, om den har kunnet genkende karakteren. Er karakteren på den nuværende markørposition således én af dem fra karaktertabel-

len, vil C-flaget blive sat, og A vil indeholde karakterkoden. Står der derimod noget andet – for eksempel grafik – der er uigenkendeligt, resættes C-flaget, og A indeholder værdien nul.

Vi har allerede i kapitel 7 berørt TXT RD CHAR-rutinen, og vi vil derfor ikke give yderligere eksempler på den her.

Den næste rutine, vi vil omtale, arbejder med vinduer. Den hedder TXT WIN ENABLE, og med den kan man sætte vinduets størrelse i den nuværende stream. Vi vil senere vende tilbage til, hvordan man kan ændre stream og derved definere vinduer helt frit.

Rutinen ligger på BB66h, og før man kalder den, skal fire registre LOAD'es med værdier:

D indeholder den ene kolonne
H indeholder den anden kolonne
E indeholder den ene række
L indeholder den anden række

Koordinaterne, som de skal indeholde, skal være angivet i forhold til skærmens øverste venstre hjørne, der har koordinaterne 0,0 og ikke som sædvanlig 1,1. Rutinen definerer ikke blot vinduet. Den tager sig – som alle andre rutiner – af sikkerhedsforanstaltninger, så det hele forløber tilfredsstillende. Efter kaldet vil markøren være placeret i vinduets øverste venstre hjørne.

Denne rutine har – som så mange andre – en parallel, nemlig den, der oplyser om det, som man her kan definere. Således findes TXT GET WINDOW, der oplyser om vinduet i den nuværende streams størrelse. Denne rutine ligger på BB69h.

Hvis du har brug for en CLS-kommando, så ligger der en rutine kaldet TXT CLEAR WINDOW på adresse BB6Ch. Den sletter skærmen på den nuværende stream. Arbejder man således uden vinduer, og dermed vel nok i stream 0, der er det normale, vil kaldet af TXT CLEAR WINDOW medføre, at hele skærmen slettes. Farven, som det sker med, er den farve, som baggrunden (PAPER) i den nuværende stream er tildelt. Desuden flyttes markøren til øverste venstre hjørne.

De næste to rutiner, som vi vil omtale, handler om skærmens farver. På tilsvarende måde til BASIC's PEN-kommando findes en maskinkoderutine til at skifte pen, og den hedder TXT SET PEN.

A-registret skal indeholde pennummeret, som man ønsker at bruge. Det er værd at bemærke, at i mode 0 må man bruge værdier i A mellem 0 og 15, i mode 1 mellem 0 og 3, og i mode 2 mellem 0 og 1.

Rutinen ligger gemt på BB90h, og AF og HL bliver ødelagt ved kaldet.

Ved siden af, på BB93h, ligger TXT GET PEN, hvor A ved returnering vil indeholde det nuværende pennummer. Begge rutiner arbejder naturligvis med den nuværende stream.

På BB96h ligger TXT SET PAPER, der skifter baggrunden ud. Forholdene er de samme for denne som for TXT SET PEN, og det samme gælder TXT GET PAPER på BB99h. Prøv selv at arbejde med disse fire rutiner, så du bliver fortrolig med dem.

På adresse BBB4h ligger en rutine, som vi allerede har nævnt findes. Den hedder TXT STR SELECT, og med den vælger man den stream, som man ønsker at arbejde med. Nummeret på den skal selvfølgelig være indeholdt i A-registret, inden rutinen kaldes.

Med denne rutine, rutinen til definition af vindue og rutinerne til pen og baggrund vil man være velbevandret, når man programmerer i maskinkode. Disse rutiner vil være særdeles velegnede, og i kapitel 21 findes flere, der udbygger dem. Vi vil senere komme med eksempler på brugen af disse rutiner og dermed vise deres styrke.

Den sidste rutine, som vi omtaler særskilt i dette kapitel, hedder TXT SET MATRIX, og med den kan man definere egne karakterer. Hvis man kalder rutinen, og man vil definere et tegn, der ikke er definerbart, sker der intet. Inden kaldet skal A indeholde nummeret på den karakter, der skal defineres, og HL skal indeholde startadressen for de otte bytes, der fortæller, hvordan karakteren skal se ud.

I starten af dette kapitel forklarede vi, hvordan en karakter var opbygget. Af dette følger, at den første værdi i tabellen skal udtrykke den første vandrette linje i den brugerdefinerbare karakter, anden værdi nummer 2 vandrette linje o.s.v.

Vi prøver nu at definere et af vore egne bogstaver, som Amstrad-producenterne i den grad har glemt: nemlig et Æ. Vi gør det med dette program:

Hex-kode	**Mnemonic**
3EFF	LD A,255
21499C	LD HL,40009
CDA8BB	CALL TXT SET MATRIX
C9	RET
3E78C8CE	DEFB
F8C8CE00	DEFB

Bogstavet ser således ud:

```
00111110 = 3E
01111000 = 78
11001000 = C8
11001110 = CE
11111000 = F8
11001000 = C8
11001110 = CE
00000000 = 00
```

CALL 40000 : PRINT CHR $ (255) vil nu give et Æ.

Du kan selv lægge den ind på tastaturet ved at bruge den rutine, som omdefinerer hvilken karakter, der hører til en given tast. Denne rutine omtalte vi ikke under vor gennemgang af tastaturet i sidste kapitel, men du kan naturligvis se den i kapitel 21.

I den store jumpblock findes en rutine, der får skærmen til at rulle op eller ned. Den hedder SCR HW ROLL og er beliggende på adresse BC4Dh. Inden man kalder rutinen, skal B-registret antage værdien nul, hvis skærmen skal rulle ned, og alt endet end nul, hvis den skal rulle op. Endvidere skal A-registret indeholde et udtryk for, med hvilken farve der skal scrolles. A skal dog ikke LOAD'es med INK-nummeret, men derimod med den specielle farvekode, som vi omtalte i starten af dette kapitel. En mere dybdegående forklaring kan du iagttage under SCR INK ENCODE i kapitel 21. Vi kan dog fortælle kort, at skal du bruge pen 0, skal A LOAD'es med 0, pen 1 kræver 240, pen 2 skal bruge 15 og pen 3 skal bruge koden 255. Bruger du andre end de fire her nævnte værdier, vil linjen blive blandet af flere farver.

Vi vil afslutte dette kapitel med et program. Det skal udskrive alle 256 karakterer i karaktersættet, men forstørret 64 gange. I LROM findes det, som kaldes karaktergeneratoren. Den ligger fra adresse 14336 til 16383, altså i alt 2 Kbytes. Karaktergeneratoren indeholder oplysninger om, hvorledes bogstaver, tal og alle de andre tegn ser ud. I karaktergeneratoren er et tegns udseende lagret i 8 bytes, eller 8 * 8 = 64 bit. Derfor fylder hele generatoren 8 * 256 = 2 Kbytes. Hvis du eksempelvis vil se, hvordan tegnet med karakterkode 70 (et 'F') ser ud, kigger du i LROM på adresse 14336 + 8 * karakterkoden. Det bliver i vores tilfælde 14336 + 8 * 70 = 14896. F ligger således fra 14896 til 14903. Prøv dette program:

Hex-kode	**Mnemonic**
CD06B9	CALL ENABLE LROM
21303A	LD HL,14896
0E08	LD C,8
L1	
7E	LD A,(HL)
0608	LD B,8
L2	
07	RLCA
F5	PUSH AF
3E30	LD A,48
CE00	ADC A,0
CD5ABB	CALL TXT OUTPUT
F1	POP AF
10F4	DJNZ L2
3E0D	LD A,13
CD5ABB	CALL TXT OUTPUT
3E0A	LD A,10
CD5ABB	CALL TXT OUTPUT
23	INC HL
0D	DEC C
20E3	JR NZ, L1
C9	RET

Her udskrives 8 binære tal. Hvis du kigger på et-tallerne, vil du kunne se, at det er et F, de danner. Vi kan illustrere det endnu bedre ved at erstatte

```
LD A,48
ADC A,0
```

med

```
LD A,32
JR NC,2
ADD A,17
```

men bemærk, at længden af rutinerne forøges med to.

Kan du se, at der er et F!

Vi skylder desuden at sige, at vi benytter to kontrolkoder i programmet, men du kan selv finde dem i appendiks E.

Med den viden kan vi nu lave det egentlige program. Vi ønsker, at programmet skal udskrive alle 256 tegn forstørret 64 gange under hinanden adskilt med et par mellemrum. Alle tegnene vil altså ikke kunne stå på skærmen på en gang, men de vil komme rullende hele tiden. Det første vi gør er at LOAD'e HL med karaktergeneratorens startadresse samt give B værdien 8, som jo er det antal bytes ét tegn fylder. Derefter gemmes de begge på stakken.

```
LD HL,14336
LØK 1
LD B,8
LØK 2
PUSH BC
PUSH HL
```

Der skal scrolles én gang, og det gøres med

```
LD B,255
XOR A
CALL SCR HW ROLL
```

B LOAD'es med 255, fordi skærmen skal rulles opad, og A nulstilles, fordi den nye linje, der kommer forneden på skærmen, skal have den samme farve som baggrunden, og vi regner ikke med at du har skiftet denne, hvis startværdi er nul. Vi skal nu skrive i række 25, kolonne 1 svarende til LOCATE 1,25. H skal altså indeholde 1 og L 25 – med andre ord skal HL ialt indeholde 1 * 256 + 25 = 281.

```
LD HL,281
CALL TXT SET CURSOR
```

Vi indsætter en test for, om DELETE-tasten er nedtrykket. Er dette ikke tilfældet, laves der et »still-billede«.

```
LD A,79
CALL KM TEST KEY
JR NZ, – 7
```

Nu kommer det centrale punkt i programmet. Vi husker, at stakken øverst har liggende startadressen for karaktergeneratoren. Den henter vi ned, og kigger samtidig i LROM på adresse HL.

```
POP HL
CALL LROM ENABLE
LD A,(HL)
```

Disse 8 bits i A skal nu behandles på samme måde, som vi gjorde det, da vi så på F's udseende. Blot vil vi udskrive kasser (CHR $ (143)) i stedet for et-taller og mellemrum (CHR $ (32)) i stedet for nuller. Den nemmeste måde er at rotere alle register A's bit ud i carry-flaget én efter én. Hvis flaget sættes, skal der udskrives en kasse, ellers et mellemrum.

```
LD B,8
LØK 3
RLCA
PUSH AF
LD A,143
JR C,2
```

```
LD A,32
CALL TXT OUTPUT
POP AF
DJNZ LØK 3
```

Som du kan se, roteres A 8 gange for at få alle 8 bit omsat til hhv. kasser og mellemrum. Ind i mellem er der PUSH'et og POP'et for at huske A's indhold, mens der udskrives. Når denne løkke er afsluttet, ligger BC, som holder styr på om tegnet er færdiglavet, stadig på stakken – vi PUSH'ede denne værdi i starten. Den henter vi nu ned, mens vi incrementerer HL for at få fat i næste byte i generatoren. Med DJNZ hoppes tilbage, hvis tegnet ikke er færdiglavet.

```
POP BC
INC HL
DJNZ LØK 2
```

Hvis der ikke springes tilbage, er tegnet altså færdiglavet. Hvis HL nu indeholder 16384 er det sidste tegn skrevet på skærmen, og vi skal returnere. Hvis HL er blevet 16384, vil bit 6 være sat.

```
BIT 6,H
RET NZ
```

Hvis der er flere tegn, kører programmet videre her, hvor vi bare indsætter en ekstra scroll for at adskille tegnene.

```
PUSH HL
LD B,255
LD A,B
CALL SCR HW ROLL
POP HL
JR LØK 1
```

Efter scroll hoppes der til LØK 1. Igen er der PUSH'et og POP'et ind imellem, men det er atter for at huske, hvor langt generatoren er nået. Scroll-rutinen lader jo ikke HL stå uberørt. At vi

LOAD'er A med 255, inden vi scroller, bevirker, at den nye linje, som bliver scrollet, bliver rød (pen 3).

Det færdige program ser med dets hex-koder tilføjet således ud:

Hex-kode	**Mnemonic**
210038	LD HL,14336
LØK 1	
0608	LD B,8
LØK 2	
C5	PUSH BC
E5	PUSH HL
06FF	LD B,255
AF	XOR A
CD4DBC	CALL SCR HW ROLL
211901	LD HL,281
CD75BB	CALL TXT SET CURSOR
3E4F	LD A,79
CD1EBB	CALL KM TEST KEY
20F9	JR NZ, – 7
E1	POP HL
CD06B9	CALL LROM ENABLE
7E	LD A,(HL)
0608	LD B,8
LØK 3	
07	RLCA
F5	PUSH AF
3E8F	LD A,143
3802	JR C,2
3E20	LD A,32
CD5ABB	CALL TXT OUTPUT
F1	POP AF
10F2	DJNZ LØK 3
C1	POP BC
23	INC HL
10D2	DJNZ LØK 2
CB74	BIT 6,H
C0	RET NZ
E5	PUSH HL

```
06FF        LD B,255
78          LD A,B
CD4DBC      CALL SCR HW ROLL
E1          POP HL
18C3        JR LØK 1
```

Kør programmet med CALL 40000 – og nyd det! Prøv at udskifte RLCA med RRCA. Det gøres med

```
POKE 40033,15
```

Prøv også at udskifte

```
JR C,2
```

med

```
JR NC,2
```

Det gøres med

```
POKE 40037,48
```

Det bevirker, at tegnene bliver inverse (omvendte).

På adresse BC50h findes en rutine, der kun scroller en del af skærmen. Find den selv i kapitel 21, og eksperimentér! Du kan i det netop forklarede program således sikre dig, at den røde tværstreg ikke fylder hele skærmens bredde. Desuden kan du lave andre forbedringer på vort program – helt efter dit eget temperament!

Kapitel 16

Vi har nu beskæftiget os med tekstskærmen, men som du formentlig allerede kender det fra BASIC, så er Amstrad også udstyret med en grafikskærm, hvorpå man kan tegne streger, plotte punkter og meget mere. Det skal vi alt sammen se på i dette kapitel, hvor vi i alt giver tre færdige programeksempler.

I den første jumpblock findes også rutiner, der er i stand til at tegne punkter, flytte pennen, tegne streger og sætte koordinatsystemets origo. De er således næsten ækvivalente med kommandoerne PLOT, MOVE, DRAW og ORIGIN. På adresse BBC0h ligger den første af dem, og den hedder GRA MOVE ABSOLUTE. Denne flytter grafikmarkørens position i x- og y-aksens retninger. Inden rutinen kaldes, skal DE indeholde X-koordinaten, og HL skal indeholde y-koordinaten. Der er her tale om absolutte koordinater, hvormed menes, at man flytter markøren til det punkt DE og HL beskriver i forhold til origo (koordinatsystemets 0,0). Origo kan man i øvrigt selv ændre. Det sker ved at kalde rutinen GRA SET ORIGIN, der ligger på adresse BBC9h. DE-registret skal indeholde origos x-koordinat i forhold til skærmens nederste, venstre punkt, og HL den tilsvarende y-koordinat. Ved start af computeren vil origo – som du sikkert allerede ved – være i nederste, venstre hjørne, der har koordinaterne (0,0). Når man kalder SCR SET MODE, som vi omtalte i sidste kapitel, vil origo blive flyttet til startværdien (0,0), så et evt. selvdefineret origo bliver spoleret.

Man kan naturligvis også plotte et punkt ind på skærmen. Rutinen til dette hedder GRA PLOT ABSOLUTE, og den er beliggende på BBEAh i jumpblock'en. Ligesom ved GRA MOVE ABSOLUTE arbejdes der i absolutte koordinater, altså i forhold til det evt. selvdefinerede origo. DE-registret skal indeholde x-koordinaten, HL-registret y-koordinaten. Naturligvis plottes punktet i den nuværende grafik ink, som ikke nødvendigvis er den samme som tekst ink.

Der findes også en rutine, der virker som en DRAW-kommando. Den hedder GRA LINE ABSOLUTE, og den virker helt som de andre, idet der benyttes DE- og HL-registret. Der tegnes følgelig med den nuværende grafik ink.

Hvis man ønsker at plotte, flytte eller tegne andre steder end i koordinatsystemets første kvadrant, så udtrykkes det negative fortegn to-komplementært. Ønskes således et plot på – 10, – 20, så skal DE indeholde 65536 – 10 = 65526 og HL 65516 (65536 – 20).

Vi har indtil nu kun set på grafik med koordinater i forhold til origo, d.v.s. med absolutte koordinater. Imidlertid findes også rutiner til brug af relative koordinater, d.v.s. koordinater set i forhold til den nuværende markørposition. Du kan selv finde GRA MOVE RELATIVE, GRA PLOT RELATIVE og GRA LINE RELATIVE i kapitel 21.

Til at vælge den rette pen og den rette baggrund findes GRA SET PEN og GRA SET PAPER. De ligger på adresse BBDEh hhv. BBE4h. A-registret skal indeholde den ønskede hhv. pen og baggrund.

Nu har vi altså lokaliseret rutiner til at vælge pen og baggrund, men de inks, som benyttes til de forskellige penne og baggrunde, er valgt af computeren ved start. I maskinkode kan vi ændre dem med rutinen SCR SET INK, der ligger på adresse BC32h. Her skal A indeholde nummeret på den ink, som skal redefineres, og B hhv. C skal indeholde numrene på de to farver, som den skal blinke imellem. Hvis man – som ved start – ikke ønsker blinkende farver, men derimod en blivende farve, skal B og C blot LOAD'es med det samme. Det tidsrum, som farverne skal blinke i, hvis man ønsker dette, kan sættes med SCR SET FLASHING, men den må du selv studere.

Vi kan således få pen 1 til at blinke mellem sort og hvidt ved at skrive

Hex-kode	**Mnemonic**
3E01	LD A,1
061A	LD B,26
0E00	LD C,0
CD32BC	CALL SCR SET INK
C9	RET

Naturligvis kan du også LOAD'e BC som et 16 bits-register, og derved spare én byte. Man skal være sparsom i disse tider . . .

Endvidere skal det nævnes, at hvis A LOAD'es med et tal større end 15, vil Amstrad selv gøre det legalt.

Vi skal nu se på et konkret eksempel på kombinering af tekstskærm og grafikskærm. Vi vil tegne et flag på skærmen. Programmet ser i sin helhed således ud:

Hex-kode	**Mnemonic**
INIT	
110000	LD DE,0
210000	LD HL,0
CDC9BB	CALL SET ORIGIN
3E01	LD A,1
011A1A	LD BC,6682
CD32BC	CALL SCR SET INK
3E02	LD A,2
010000	LD BC,0
CD32BC	CALL SCR SET INK
3E02	LD A,2
CDDEBB	CALL SET GRA PEN
STANG	
114808	LD DE,8 * 256 + 72
L1	
211400	LD HL,20
D5	PUSH DE
1600	LD D,0
CDC0BB	CALL GRA MOVE ABSOLUTE
110000	LD DE,0
215E01	LD HL,350
CDF9BB	CALL GRA LINE RELATIVE
D1	POP DE
1C	INC E
15	DEC D
20E9	JR NZ,L1
FLAG	
21A59C	LD HL,40101
0606	LD B,6

```
L2
C5          PUSH BC
7E          LD A,(HL)
23          INC HL
46          LD B,(HL)
23          INC HL
56          LD D,(HL)
23          INC HL
4E          LD C,(HL)
23          INC HL
5E          LD E,(HL)
23          INC HL
E5          PUSH HL
60          LD H,B
69          LD L,C
CD44BC      CALL SCR FILL BOX
E1          POP HL
C1          POP BC
10EB        DJNZ L2
SCROLL
0610        LD B,16
L3
C5          PUSH BC
AF          XOR A
06FF        LD B,255
210205      LD HL,1282
11160D      LD DE,3350
CD50BC      CALL SCR SW ROLL
C1          POP BC
10F0        DJNZ L3
C9          RET
DATA
FF05071213  DEFB
FF05071516  DEFB
FF090D1213  DEFB
FF090D1516  DEFB
F0050D1414  DEFB
F008081216  DEFB
```

Programmet består af flere forskellige rutiner, der har hvert sit formål. Vi vil her gøre rede for tankegangen i rutinerne uden at gå i detaljer, men det er vigtigt, at du hele tiden forstår de skridt, vi tager. Desuden har vi ikke valgt den billigste løsning, men en løsning, som er nemmest at forstå.

Programmet indledes med en initialisering af skærmen og dens farver. Først så sættes origo ved hjælp af SCR SET ORIGIN, og der ændres på de bestående inks. Farvenummer 1 ændres til hvid, og farvenummer 2 ændres til sort. Vi regner med, at du arbejder i mode 1, så farvenummer 0 er fortsat blå og farvenummer 3 rød. Desuden skiftes den aktuelle pen (den grafiske pen) til pen 2, der er sort.

Så kommer rutinen STANG, der tegner flagstangen. Denne består af 8 lodrette, sorte streger, der alle har samme højde. Først LOAD'es DE med 8 * 256 + 72. DE-registret skal i denne rutine betragtes som to 8 bits-registre, der har hver sin opgave. D varetager antallet af gennemløb i rutinen (i alt 8), mens E tager sig af x-koordinaten til flagstangen. Først i L1 flyttes grafikpositionen hen til E,20 – hvor E først antager 72. Dernæst tegnes relativt opad, hvormed som nævnt tidligere menes, at der tegnes i forhold til den nuværende grafikposition. Da flagstangen er helt lodret, tegnes 0,350. Undervejs er DE lagt op på stakken for at holde styr på tællerne, og den POP'es nu tilbage. Så incrementeres E, så den næste gang rykker grafikpositionen én længere til højre. D decrementeres for at undersøge, om der er flere gennemløb. Er dette tilfældet, gennemløbes L1 igen.

Hvis det derimod ikke er tilfældet, gås videre til rutinen FLAG, der ikke indeholder grafik, men derimod bruger tekstskærmen. Den benytter således rutinen SCR FILL BOX, hvormed en del af skærmen kan farvelægges med en i A-registret givet farve. A skal dog ikke indeholde ink-nummeret, men derimod en farvekode som beskrevet i forrige kapitel. Når SCR FILL BOX kaldes, skal H og D angive de to grænseflader for farvelægningen horisontalt, mens L og E angiver det vertikalt. Bagest i programmet er skrevet en tabel, der indeholder 30 værdier. Rutinen FLAG skal lave 6 farvelægninger, så i alt er der 5 data pr. farvelægning. I selve rutinen bliver HL først LOAD'et med startadressen for alle disse data, og B LOAD'es med antallet af farvelægninger, hvorefter dennes værdier lægges på stakken. Så hentes værdierne ind fra

datablokken én efter én, i alt fem. Den første er farvekoden, de fire næste grænserne for farvelægningen. Så lagres værdien af tælleren HL. Da vi hentede værdierne, lagdes HL's værdier i BC, men disse kan nu overføres til HL, inden SCR FILL BOX kaldes. Så hentes adressen på det næste data og tælleren tilbage, og er der flere data gennemløbes L2 igen.

Den sidste rutine omtalte vi flygtigt i slutningen af sidste kapitel, og det er den, der kan få skærmen til at rulle. Denne skal i alt rulle 16 gange, hvorfor B er LOAD'et med dette tal. Når SCR SW ROLL kaldes, skal A indeholde den farvekode, som den nye linje på skærmen skal indtage, og det er 0 for ink 0 (blå). B skal indeholde en værdi til oplysning, om der skal rulles op eller ned. Hvis B ikke er nul, rulles opad, og vi har valgt at LOAD'e B med 255. DE og HL skal indeholde værdierne for det »vindue«, der skal scrolles.

Som allerede nævnt i starten kan dette program effektiviseres. F.eks. kan du fjerne instruktionen

```
LD B,255
```

i SCROLL-rutinen! Kan du se hvorfor? B indeholder alligevel altid noget forskelligt fra nul, undtagen når den er talt helt ned ved DJNZ-instruktionen, og så returneres der. For SCR SW ROLL er det nok, at B bare er forskellig fra nul, hvis den skal scrolle opad. Prøv selv at finde andre muligheder til besparelse.

Man kan også i maskinkode definere sit eget grafikvindue. Til dette formål skal bruges to rutiner, hvor den ene angiver beliggenheden i x-aksens retning, mens den anden gør det i y-aksens. Rutinerne hedder GRA WIN WIDTH og GRA WIN HEIGHT, der ligger på hhv. BBCFh og BBD2h. Med GRA WIN WIDTH bestemmes vinduets position i bredden. DE-registret angiver afstanden fra skærmens venstre kant til vinduets ene kant, mens HL-registret indeholder afstanden til den anden kant. Bemærk, at koordinaterne her angives som standard koordinater, d.v.s. i forhold til skærmens nederste, venstre hjørne og *ikke* i forhold til et selvdefineret origo. Skærmens venstre kant vil således fungere som y-akse.

Indeholder DE 100 og HL 300, så vil vinduet være beliggende i en bredde af 200 punkter fra x-koordinaten 100 til 300.

GRA WIN HEIGHT definerer på samme måde højden, og DE-registret skal indeholde den ene afstand fra skærmens nederste kant (der i standard koordinater jo fungerer som x-akse), mens HL indeholder den anden afstand.

Bemærk i øvrigt, at hvis man bruger rutiner, der tegner linjer eller plotter punkter, og dette medfører brug af et punkt udenfor de definerede vinduer, så bliver punktet ikke berørt. Grafikmarkøren derimod bliver stående udenfor vinduets kant.

Naturligvis findes der også rutiner, der aflæser et vindues størrelse, men dem må du selv finde i kapitel 21.

På BBDBh ligger rutinen GRA CLEAR WINDOW, der clear'er et vindue. Det gøres med den nuværende grafik ink.

Bemærk også, at har man ikke defineret noget vindue, vil Amstrad automatisk opfatte hele skærmen som et stort vindue, og kald af GRA CLEAR WINDOW, vil da clear'e hele skærmen.

Den sidste rutine, som vi vil omtale, hedder GRA WR CHAR, og med den kan man udskrive en karakter på den nuværende grafikmarkørs position. A-registret skal LOAD'es med koden for karakteren, og så klarer rutinen selv resten. Helt præcist sagt bliver bogstavets øverste venstre hjørne skrevet præcist på grafikpositionen, og når man har udskrevet karakteren, vil grafikmarkørens position have rykket sig. Da en karakter i mode 2 fylder otte punkter i bredden, vil grafikpositionen i denne tilstand flyttes 8 punkter til højre, hvorimod der naturligvis ingen ændring er i y-aksens retning. I mode 1 vil den blive rykket 16 punkter, mens den i mode 0 vil blive flyttet 32.

Karakteren, som udskrives, vil blive skrevet med den nuværende *grafik* pen, og baggrunden vil blive den nuværende *grafik* paper.

Vi vil afslutte dette kapitel med to store programmer. Dette første skal tegne søjlediagrammer for os. Vi kan forestille os et firma, der har et årligt salg fordelt på de 12 måneder, der ser således ud:

Januar:	36 mill. kr.
Februar:	44 mill. kr.
Marts:	67 mill. kr.
April:	72 mill. kr.
Maj:	51 mill. kr.
Juni:	41 mill. kr.
Juli:	25 mill. kr.
August:	56 mill. kr.
September:	79 mill. kr.
Oktober:	74 mill. kr.
November:	66 mill. kr.
December:	86 mill. kr.

Søjlediagrammet skal være aftegnet i koordinatsystem, og desuden have indlagt en vis dybdevirking. Koordinatsystemet skal have afsat månederne ud ad abscissen (x-aksen), og salget i de enkelte måneder op ad ordinaten (y-aksen). Da vi har bygget programmet op omkring ovenstående fiktive firmas salg, så vil det maksimale salgsbeløb i en enkelt måned være 90 mill. kr. På y-aksen angives tallet i hele millioner.

Programmet ser i sin helhed således ud:

Hex-kode	**Mnemonic**
INIT	
113200	LD DE,50
62	LD H,D
6B	LD L,E
CDC9BB	CALL GRA SET ORIGIN
3E02	LD A,2
010000	LD BC,0
C5	PUSH BC
C5	PUSH BC
C5	PUSH BC
C5	PUSH BC
CD32BC	CALL SCR SET INK
3E01	LD A,1
CDDEBB	CALL SET GRA PEN
AKSER	
114902	LD DE,585

```
E1              POP HL
CDF6BB          CALL GRA LINE ABSOLUTE
D1              POP DE
E1              POP HL
CDC0BB          CALL GRA MOVE ABSOLUTE
D1              POP DE
212C01          LD HL,300
CDF6BB          CALL GRA LINE ABSOLUTE
BOGSTAV
DD212F9D        LD IX,40239
110D00          LD DE,13
060C            LD B,12
L1
C5              PUSH BC
D5              PUSH DE
21F6FF          LD HL,-10
CDC0BB          CALL GRA ABSOLUTE
DD7E00          LD A,(IX+0)
DD23            INC IX
CDFCBB          CALL GRA WR CHAR
D1              POP DE
213100          LD HL,49
19              ADD HL,DE
EB              EX DE,HL
C1              POP BC
10E7            DJNZ L1
VÆRDI
210400          LD HL,4
01300A          LD BC,10*256+48
L2
E5              PUSH HL
C5              PUSH BC
11D8FF          LD DE,-40
CDC0BB          CALL GRA MOVE ABSOLUTE
C1              POP BC
C5              PUSH BC
79              LD A,C
FE30            CP 48
2002            JR NZ,2
```

```
3E20        LD A,32
CDFCBB      CALL GRA WR CHAR
3E30        LD A,48
CDFCBB      CALL GRA WR CHAR
C1          POP BC
0C          INC C
E1          POP HL
111E00      LD DE,30
19          ADD HL,DE
10DE        DJNZ L2
SØJLE 1
110A00      LD DE,10
060C        LD B,12
L3
C5          PUSH BC
060B        LD B,11
L4
C5          PUSH BC
D5          PUSH DE
210200      LD HL,2
CDC0BB      CALL GRA MOVE ABSOLUTE
3E02        LD A,2
CDDEBB      CALL GRA SET PEN
DD6E00      LD L,(IX + 0)
2600        LD H,0
E5          PUSH HL
29          ADD HL,HL
D1          POP DE
19          ADD HL,DE
110000      LD DE,0
CDF9BB      CALL GRA LINE RELATIVE
3E01        LD A,1
CDDEBB      CALL GRA SET PEN
110800      LD DE,8
211400      LD HL,20
CDF9BB      CALL GRA LINE RELATIVE
D1          POP DE
13          INC DE
13          INC DE
```

```
C1          POP BC
10D0        DJNZ L4
SØJLE 2
3E02        LD A,2
322E9D      LD (40238),A
0604        LD B,4
L5
C5          PUSH BC
D5          PUSH DE
210200      LD HL,2
CDC0BB      CALL GRA MOVE ABSOLUTE
3E03        LD A,3
CDDEBB      CALL GRA SET PEN
3A2E9D      LD A,(40238)
DD6E00      LD L,(IX + 0)
2600        LD H,0
E5          PUSH HL
29          ADD HL,HL
D1          POP DE
19          ADD HL,DE
4F          LD C,A
0600        LD B,0
09          ADD HL,BC
C606        ADD A,6
322E9D      LD (40238),A
110000      LD DE,0
CDF9BB      CALL GRA LINE RELATIVE
D1          POP DE
13          INC DE
13          INC DE
C1          POP BC
10D2        DJNZ L5
NY SØJLE
211300      LD HL,19
19          ADD HL,DE
EB          EX DE,HL
DD23        INC IX
C1          POP BC
108E        DJNZ L3
```

```
C9              RET
DATA
00              DEFB – ADDITION –
4A464D41        DEFB – BOGSTAV –
4D4A4A41        DEFB
534F4E44        DEFB
242C4348        DEFB – VÆRDI –
32291938        DEFB
4F4A4256        DEFB
```

Prøv at indtaste og køre programmet. Så vil du få nemmere ved at forstå vor forklaring, der nu følger. Programmet *skal* ligge fra adresse 40000, og du skal køre i mode 1.

Først ligger rutinen INIT, der initialiserer skærmen. Først sættes origo til 50,50, og ink 2 bliver sort. Bemærk, at der er placeret 4 PUSH-instruktioner, men dette er kun for at spare bytes. Vi henter dem nemlig hurtigt tilbage. Dernæst sættes den nuværende grafiske pen til at være nummer 1.

Så kommer rutinen AKSER, der tegner koordinatsystemets akser. Først tegnes abscissen, hvorefter der flyttes tilbage til origo, og ordinataksen tegnes.

I adresse 40044 starter rutinen BOGSTAV, der udskriver forbogstaverne på månederne: disse skal stå under x-aksen. IX peger på det første bogstav, der står i datablokken sidst i programmet. DE LOAD'es med 13, der er startværdien for, hvor langt ude ad x-aksen bogstavet skal skrives. Da der er 12 bogstaver, LOAD'es B med dette tal og fungerer atter som tæller. Så starter selve løkken, og først så gemmes BC og DE. HL LOAD'es med 65526 (– 10), hvorefter grafikpositionen flyttes ved kald af GRA MOVE ABSOLUTE. Første gang vil der altså blive flyttet til 13, – 10 (DE,HL), og i slutningen af løkken bliver DE så tillagt en værdi for at nå længere ud ad x-aksen. Efter flytningen hentes bogstavets karakterkode ind i A, og IX incrementeres, så den peger på næste karakter. Så skal karakteren udskrives, og det sker med GRA WR CHAR, der skriver den på den nuværende grafikposition. Så hentes DE tilbage, og der lægges 49 til. Det sker således:

```
POP DE
LD HL,49
```

```
ADD HL,DE
EX DE,HL
```

Da vi kun kan addere DE med HL-registret (IX eller IY kan i teorien også anvendes), så LOAD'es HL med det tal, der skal adderes, og de lægges sammen. Imidlertid skal resultatet stå i DE og ikke i HL, så de ombyttes med EX DE,HL. Her får DE altså resultatet, mens HL får DE's værdi, men denne er ubetydelig: vi skal ikke bruge den. Så hentes tælleren tilbage fra stakken, og der gennemløbes igen, hvis B ikke blev nul.

Den næste rutine, på adresse 40078, udskriver værdierne op ad y-aksen. Vi skal have udskrevet værdierne 0, 10, 20, 30, . . . , 80 og 90. Dette kan nemmest gøres på den måde, at en tæller holder styr på 1. ciffer. Det andet ciffer er jo altid nul. Disse to cifre udskrives så hver for sig, og efter udskriften incrementeres tælleren for at kende næste værdi.

Vi starter med at LOAD'e HL med 4. Det er som ved BOGSTAV en startværdi, men denne gang for bevægelsen op ad y-aksen. Før var det grafikpositionen på x-aksen, der reguleredes, denne gang er det på y-aksen, og derfor er det HL, der skal ændres sidst i løkken. BC fungerer som to 8 bits-registre, hvor B LOAD'es med 10 – antallet af værdier. C LOAD'es med 48, der er karakterkoden for nul, den første værdi. Så starter selve løkken, og der gemmes HL og BC. DE LOAD'es med 65496 (– 40), hvorefter GRA MOVE ABSOLUTE kaldes. Første gang vil grafikpositionen blive rykket til (– 40,4). Så skal vi have hentet værdien i C tilbage, og derfor POP'er og PUSH'er vi BC (BC skal stadig gemmes på stakken). Så LOAD'es A med værdien i C, og der sker en lille sammenligning. Er A 48 (karakterkoden for nul), skal den i stedet for være 32, der er et mellemrum. Det skyldes, at nul ikke skal udskrives som 1. ciffer, da 2. ciffer altid er et nul. Vi skal jo ikke have udskriften 00! Den første karakter udskrives dernæst – det er et mellemrum, hvis 0 er 1. ciffer – og dernæst udskrives et nul. Så hentes BC tilbage, og C incrementeres for at finde næste værdi. HL hentes tilbage, og den tillægges 30, inden der gennemløbes pany, hvis B ikke er blevet nul. At der lægges 30 til betyder, at der forrest i løkken ved 2. gennemløb bliver flyttet grafikposition hen til (– 40,4 + 30) = (– 40,34).

Så kommer selve tegningen af søjlerne. De er opbygget på

den måde, at selve søjlen består af 11 helt tætliggende streger i samme farve. Da disse i mode 1 fylder 2 punkter hver horisontalt, vil en søjle fylde 22 punkter ud af de 640. Dernæst – efter hver enkelt lodret streg – laves en skrå streg, der skal illudere toppen. Herefter laves 4 streger i samme farve, der skal illudere dybdevirkningen. Ialt fylder søjlen altså (11 + 4) * 2 = 30 punkter i x-aksens retning. Mellem hver søjle er der 19 punkters mellemrum.

Først LOAD'es DE med 10, da der ved start ikke skal laves en søjle helt til venstre ved y-aksen, men derimod først 10 punkter inde. Så LOAD'es B med antallet af søjler, 12, og denne værdi gemmes på stakken. B LOAD'es nu om igen med 11, der er det nævnte antal af lodrette streger i den egentlige søjle. Denne værdi og DE's værdi gemmes, hvorefter HL LOAD'es med 2, da alle søjler skal starte to punkter over x-aksen. Med kald af GRA MOVE ABSOLUTE vil grafikpositionen første gang blive rykket til (10,2). Så sættes pennens farve til sort (pen 2). Da VÆRDI var færdig, pegede IX på den første værdi efter det sidste bogstav, og denne værdi er ikke ændret. I datablokken kan du se, at her ligger vore værdier, der skal vise, hvor høje søjlerne er. Da disse værdier ligger mellem 0 og 100, fylder de kun én byte hver. For at blive afsat på y-aksen, skal de multipliceres med 3 (herved fås en højde mellem 0 og 300 punkter). Dette tal skal ligge i HL. Derfor starter vi med at hente værdien mellem 0 og 100 ind i L-registret, og MSB af HL-registret, H, nulstilles. Så multiplicerer vi HL med 3. Det sker således:

```
PUSH HL
ADD HL,HL
POP DE
ADD HL,DE
```

Nu ligger højden af søjlen i form af antal punkter i HL, og da stregen skal være helt lodret, LOAD'es DE med nul. Så tegnes relativt, d.v.s. ud fra det sted, hvor den nuværende grafikposition er. Den blev tidligere sat ved kald af GRA MOVE ABSOLUTE. Så skiftes pen til nummer 1 (gul), og der tegnes relativt (8,20) fra toppen af den afsatte streg. Herved fås en skrå streg ovenpå den lodrette. Så hentes DE tilbage, og vi skal nu gøre klar til en ny

streg. Den starter to punkter længere til højre ad x-aksen, så vi incrementerer DE to gange. Desuden hentes BC tilbage for at undersøge, om der skal tegnes flere lodrette streger. Er dette tilfældet, gås tilbage, ellers står vi nu klar til at lave fire streger med dybdevirkning.

Disse streger skal imidlertid ikke have samme højde. Den første streg skal være 2 punkter længere end de netop afsatte lodrette streger, den næste skal være 8 længere, den tredje 14 længere og den sidste hele 20 punkter længere. Derfor LOAD'es adresse 40238 med det tal, der skal adderes søjlens rigtige højde. Dette tal vil første gang være 2, så vi lægger dette tal ind på adressen. Dernæst LOAD'es B med 4, der jo er antallet af streger. BC og DE PUSH'es. DE indeholder stadigvæk stedet på x-aksen, hvor den næste streg skal udgå fra. HL LOAD'es med 2, hvorefter der flyttes til denne position med GRA MOVE ABSOLUTE. Så skiftes pen til rød (nummer 3), og vi henter atter søjlens højde ind fra datablokken. Husk, at IX ikke er blevet ændret. Denne værdi multipliceres som allerede nævnt med 3, og vi tillægger additionsfaktoren lagret på adresse 40238. Inden multiplikationen har vi hentet denne værdi ind i A, og den overføres nu til C, og B-registret LOAD'es med nul. BC kan nu tillægges HL. Så skal additionsfaktoren forøges med 6, og vi er da klar til at foretage tegningen af den lodrette streg. DE skal først LOAD'es med nul, da den er helt lodret – HL indeholder højden – og vi kalder GRA LINE RELATIVE. Så gøres DE to større for at være klar til næste streg. BC hentes tilbage, og de gennemløbes atter, hvis B ikke er blevet nul.

Nu når vi til rutinens sidste del. Her skal vi have lavet mellemrum mellem hver enkelt søjle, og derfor skal DE adderes med 19 – antallet af punkter mellem hver søjle, der således i alt fylder 49 punkter. BC hentes ligeledes tilbage, og den var jo i starten af SØJLE 1-løkken PUSH'et. Den indeholder antallet af søjler, og derfor gennemløbes pøny, hvis B ikke er blevet nul. Hvis dette er tilfældet, returneres der til BASIC.

Vi har nu gennemgået programmet, men vi råder dig til at forstå det til dets mindste detalje. Det vil give dig øvelse og forståelse for maskinkodeprogrammering, og det er nødvendigt, inden du selv kaster dig ud i større eksempler.

Vort sidste program benytter ingen ROM-rutiner. Dets formål er at scrolle skærmen. Det skal imidlertid ikke scrolle en hel række, men kun 1/8 af den, svarende til én vandret linje. Der er altså tale om højopløsnings-scroll.

Vi rekapitulerer skærmbilledets opbygning:

1. rk. – 1. linje: 49152, 49153, 49154, . . . , 49230, 49231
1. rk. – 2. linje: 51200, 51201, 51202, . . . , 51278, 51279
1. rk. – 3. linje: 53248, 53249, 53250, . . . , 53326, 53327
1. rk. – 4. linje: 55296, 55297, 55298, . . . , 55374, 55375
1. rk. – 5. linje: 57344, 57345, 57346, . . . , 57422, 57423
1. rk. – 6. linje: 59392, 59393, 59394, . . . , 59470, 59471
1. rk. – 7. linje: 61440, 61441, 61442, . . . , 61518, 61519
1. rk. – 8. linje: 63488, 63489, 63490, . . . , 63566, 63567

2. rk. – 1. linje: 49232, 49233, 49234, . . . , 49310, 49311
2. rk. – 2. linje: 51280, 51281, 51282, . . . , 51358, 51359

Sådan ser skærmbilledet ud, og dets specielle opbygning komplicerer også vort program. Det, der skal ske i programmet, er, at alle linjer rykkes én op. 2. linje rykkes til 1. linje, 3. linje til 2. linje o.s.v. Efter flytning af 8. linje, skal 2. rækkes 1. linje rykkes til 1. rækkes 8. linje.

Vi glemmer et øjeblik hele den første række, da denne skal behandles lidt specielt. Tilbage står 24 rækker à 8 linjer. Den øverste linje i rækken skal rykkes op til den nederste i rækken ovenover, hvorefter de 7 sidste linjer i rækken blot skal rykkes op. Vi skal bruge en underrutine, der flytter disse 7 linjer. Den ser således ud:

```
LD A,7
NÆSTE
LD BC,1968
ADD HL,BC
PUSH HL
LD BC,80
LDIR
POP DE
```

```
DEC A
JR NZ, NÆSTE
RET
```

Inden rutinen kaldes fra et hovedprogram, som vi udvikler om lidt, skal DE indeholde adressen på den pågældende rækkes 1. linje, mens HL skal indeholde adressen på rækkens 2. linje fratrukket 1968! Det skyldes, at vi i starten af rutinen lægger 1968 til HL, inden LDIR udføres. Dette gøres, fordi der imellem hver linje i en række netop er 2048 bytes, og vor LDIR-instruktion adderer 80 gange. Herved mangler netop 1968 i at nå den næste linje i rækken.

Underrutinen starter med at LOAD'e A med 7, da der skal rykkes 7 linjer, inden der returneres til hovedprogrammet. Herefter tillægges HL 1968 som allerede nævnt, og HL gemmes på stakken. BC LOAD'es med antallet af bytes i en linje, og det er 80. Derefter rykkes linjen én op med LDIR-instruktionen. Nu henter vi vort tal ned fra stakken – men det lægges i DE! Hvis vi ser på underrutinen behandlet med 2. række som eksempel, så vil DE inden rutinens kald indeholde adressen på 1. linje, mens HL efter tillægningen indeholder adressen på 2. linje. Så gemmes denne værdi, og LDIR udføres. Nu skal DE for at være klar til næste flytning indeholde adressen på 2. linje, og det var netop den, vi lagrede på stakken. Derfor hentes den ned i DE-registret. HL er ved LDIR blevet tillagt 80, men for at registret kan pege på 3. linje mangler jo netop addition med 1968, og det er den, der foretages, når der hoppes tilbage, hvis A ikke er blevet nul. Hvis alle 7 linjer er rykket, returneres.

Vi har fået forklaret den lidt besværlige og komplicerede underrutine, og vi skal nu lave hovedprogrammet. Her skal startes med første række, og den skal have rykket linjerne 2-8. Inden rutinen, som vi forklarede før, kaldes, skal DE og HL jo tillægges startværdier.

```
LD HL,49232
PUSH HL
LD DE,49152
CALL UNDERRUTINE
```

HL og DE LOAD'es med hhv. 49232 og 49152. Sidstnævnte værdi er placering af 1. rækkes 1. linje, mens 49232 er den 2. linje i denne række fratrukket 1968, som allerede forklaret.

Nu står vi klar til at flytte 2. rækkes 1. linje til 1. rækkes 8. linje. Hvad er startadressen på den førstnævnte? Det er 49232, og den PUSH'ede vi derfor før. Den hentes ned nu, og DE LOAD'es med startadressen for den sidste (8.) linje i første række.

```
POP HL
LD DE,63488
```

Nu skal vi have lavet en løkkestruktur, da der jo i alt skal foregå 24 rækkers flytning. B LOAD'es med dette tal, og værdien PUSH'es.

```
LD B,24
IGEN
PUSH BC
```

Nu skal vi have flyttet den øverste linje i en række til den nederste linje i den ovenover. DE og HL er allerede initialiseret til den første flytning, så der resterer kun at give BC værdien 80 – antallet af bytes, der skal flyttes.

```
LD BC,80
```

Nu skal vi have gemt DE og HL igen. DE indeholder startadressen på den nederste linje i rækken ovenover, mens HL indeholder adressen på den øverste linje i den nuværende række.

```
PUSH HL
PUSH DE
PUSH HL
LDIR
```

LDIR varetager flytningen af den første linje i en række til den nederste i den ovenover. Det har vi allerede beskrevet grundigt. Nu står vi klar til at flytte de næste syv linjer i rækken. På nuvæ-

rende tidspunkt indeholder HL adressen på den 1. linje + 80 (disse blev tillagt ved LDIR). For at registret kan pege på den 2. linje, som det skal, skal altså tillægges yderligere 1968, men dette klarer underrutinen jo! Så mangler vi at LOAD'e DE, og det skal være med startadressen for den 1. linje. Den gemte vi øverst på stakken.

```
POP DE
CALL UNDERRUTINE
```

Tilbage på stakken ligger to værdier – foruden rækketælleren i register B. Øverst ligger adressen på den sidste linje i rækken ovenover. Hvis vi tillægger denne 80, vil værdien være adressen på den nuværende rækkes sidste linje. Denne skal vi jo netop bruge, da vi – når vi løber tilbage i løkken – står klar til at overføre fra en rækkes øverste linje til den nederste i den ovenover. Værdien skal ligge i DE, så vi gør følgende:

```
POP HL
LD BC,80
ADD HL,BC
EX DE,HL
```

Tilbage på stakken – foruden tælleren – ligger startadressen på den øverste linje i den nuværende række. Hvis denne tillægges 80, vil vi få adressen for den øverste linje i den kommende række, og denne skal vi jo netop bruge. Herved er både DE og HL klar til at flytte.

```
POP HL
ADD HL,BC
```

Vi rekapitulerer: i øjeblikket indeholder HL adressen på den øverste linje i en række, mens DE indeholder den for den nederste linje i rækken ovenover. Vi er altså således klar til at gentage vor flytning fra starten af rutinen, så vi undersøger, om alle rækker er flyttet:

```
POP BC
DJNZ IGEN
```

Hvis der ikke er flere rækker, hoppes der ikke tilbage, men der fortsættes med dette:

```
LD HL,65408
LD B,80
LD (HL),0
INC HL
DJNZ – 5
RET
```

HL LOAD'es med startadressen for række 25s nederste linje. I denne linje er jo – som i alle øvrige – 80 bytes, og disse tildeles værdien 0. Dette betyder, at de får alle baggrundsfarven, forudsat denne er »PAPER 0« naturligvis. Disse bytes slettes altså.

Det samlede program ser således ud, og det skal ligge fra adresse 40000, ellers må du ændre CALL-adresserne:

Hex-kode	**Mnemonic**
2150C0	LD HL,49232
E5	PUSH HL
1100C0	LD DE,49152
CD739C	CALL UNDERRUTINE
E1	POP HL
1100F8	LD DE,63488
0618	LD B,24
IGEN	
C5	PUSH BC
015000	LD BC,80
E5	PUSH HL
D5	PUSH DE
E5	PUSH HL
EDB0	LDIR
D1	POP DE
CD739C	CALL UNDERRUTINE
E1	POP HL

```
015000          LD BC,80
09              ADD HL,BC
EB              EX DE,HL
E1              POP HL
09              ADD HL,BC
C1              POP BC
10E8            DJNZ IGEN
2180FF          LD HL,65408
0650            LD B,80
3600            LD (HL),0
23              INC HL
10FB            DJNZ – 5
C9              RET
UNDERRUTINE
3E07            LD A,7
NÆSTE
01B007          LD BC,1968
09              ADD HL,BC
E5              PUSH HL
015000          LD BC,80
EDB0            LDIR
D1              POP DE
3D              DEC A
20F2            JR NZ, NÆSTE
C9              RET
```

Inden programmet afprøves, skal vi lige fortælle dig, at skærmens øverste, venstre hjørne skal have adresse 49152. Dette kan du – som vi allerede har omtalt ved andre programmer – sikre med en MODE-kommando.

Sørg for, at du har et program liggende i computeren, for eksempel hex-loaderen. Eksekvér da

```
MODE 1 : LIST
FOR N = 1 TO 64 : CALL 40000 : NEXT
```

Som du vil se, vil der i alt blive scrollet 8 rækker. De 64 er jo antallet af vandrette linjer, og der er 8 linjer på en række. Et enkelt kald vil således medføre scroll af en enkelt vandret linje.

Dette program er blevet lavet uden hjælp fra Amstrads ROM, og det er måske nok noget klodset udformet. Det kan dog gøres meget mere overskueligt og enkelt. Til dette må du slå op i kapitel 21 og finde rutinen: SCR NEXT LINE (&BC26). Lav selv denne rutine – det burde ikke volde meget besvær og er desuden en glimrende øvelse!

Kapitel 17

Amstrad har indbygget en 3-kanals lydgenerator over 8 oktaver. Vi vil nu se lidt nærmere på, hvorledes denne kan styres via de rutiner, der er i operativsystemet. Til sidst i kapitlet vil vi give et færdigt eksempel.

Den første rutine hedder SOUND QUEUE, og den er faktisk analog til en SOUND-kommando i BASIC. Den ligger på adresse BCAAh.

Den pågældende lyd lægges på lydkøen for de pågældende kanaler. Er en sådan optaget, vil lyden ikke blive overført, men ignoreret. Inden rutinen kaldes, skal HL-registret indeholde adressen på en lydblok på 9 bytes. Blokken ser således ud:

Byte 1: Kanalstatus
Byte 2: Lydstyrkeændring (volume envelope)
Byte 3: Toneændring (tone envelope)
Byte 4-5: Tone
Byte 6: Støjperiode
Byte 7: Lydstyrke
Byte 8-9: Tonens varighed

Den første byte står altså for kanalstatus, og her har de enkelte bits betydning.

bit 0 sat: sender en lyd til kanal A
bit 1 sat: sender en lyd til kanal B
bit 2 sat: sender en lyd til kanal C
bit 3 sat: rendez-vous med kanal A
bit 4 sat: rendez-vous med kanal B
bit 5 sat: rendez-vous med kanal C
bit 6 sat: stopper lyd
bit 7 sat: tømmer lydkøen

Rendez-vous bruges til synkronisering af toner fra flere kanaler.

Den anden byte angiver volume envelope-nummeret, og det er derfor et tal mellem 0 og 15. Uden nogen lydstyrkeændring angives 0.

Den tredje byte angiver tone envelope-nummeret, og det er derfor også et tal mellem 0 og 15, hvor 0 angiver, at der ingen toneændring er.

De to næste bytes i blokken angiver tonen, og det er et tal mellem 0 og 4096 (2^{12}).

Støjperioden angiver, hvor meget støj der skal tilføjes lyden. Det er et tal mellem 0 og 31, hvor 0 betyder, at lyden ikke efterfølges af støj.

Lydstyrken er et tal mellem 0 og 15, hvor 15 er den maksimale lydstyrke.

Tonens varighed skal i praksis ligge mellem 0 og 32767. Den skal angives i hundrededele sekunder. En tone kan altså i alt højst ved brug af en enkelt ordre vare ca. 5 1/2 minut.

SOUND QUEUE har en ekstra lille finesse. Når en lyd skal lægges på lydkøen, så kan der være optaget; forstået på den måde, at ikke alle lyde er blevet udført endnu. Der kan på en lydkø kun stå fem lyde, nemlig den, der udføres, og fire, der venter. Kalder man således SOUND QUEUE, og der ikke er plads på køen, så vil C-flaget blive resat – og naturligvis sat, hvis alt går planmæssigt.

Der findes også en rutine, der svarer til ENV i BASIC. Den ligger på adresse BDBCh, og den hedder SOUND AMPL ENVELOPE. A-registret skal indeholde envelope-nummeret, og HL skal indeholde startadressen på en blok, som beskriver lydstyrkeændringen. Denne datablok er ikke i samme størrelse altid. Denne varierer efter antallet af sektioner i envelope'n. Som du formentlig ved fra den medfølgende instruktionsbog, kan der ialt være op til fem sektioner.

Den første byte i datablokken angiver, hvor mange sektioner der er. Tallet er altså mellem 0 og 5.

Byte 2 - 4: 1. sektion af envelope'n
Byte 5 - 7: 2. sektion
Byte 8 - 10: 3. sektion
Byte 11 - 13: 4. sektion
Byte 14 - 16: 5. sektion

Hvis der er angivet 0 sektioner, vil der blive holdt en konstant lydstyrke i 2 sekunder.

Af ovenstående ses også, at hvis der f.eks. er 3 sektioner i envelope'n, så vil hele blokken fylde 10 bytes. De tre bytes i den enkelte sektion angiver hhv. antallet af ændringer, ændringens størrelse og hvor lang tid der går fra en ændring til den næste. Eksempel:

Hex-kode	**Mnemonic**
3E01	LD A,1
214F9C	LD HL,40015
CDBCBC	CALL SOUND AMPL ENVELOPE
21569C	LD HL,40022
CDAABC	CALL SOUND QUEUE
C9	RET
02	DEFB – ANTAL SEKTIONER –
050214	DEFB – SEKTION 1 –
05FE14	DEFB – SEKTION 2 –
01	DEFB – KANAL –
01	DEFB – LYDSTYRKEÆNDRING - NUMMER
00	DEFB – TONEÆNDRING – NUMMER
C800	DEFB – TONE –
00	DEFB – STØJPERIODE –
01	DEFB – STARTLYDSTYRKE –
C800	DEFB – VARIGHED –

Den første envelope-sektion består af de tre parametre 5, 2 og 20, og den anden sektion består af 5, – 2 og 20. Læg mærke til, at de – 2 er angivet to-komplementært, altså ved 254. Den første af de to sektioner betyder, at lydstyrken skal hæves med 2 ialt 5 gange, og den tid, der går imellem ændringerne, er 20/100 (0,2) sekund. Næste sektion bevirker, at lydstyrken sænkes med 2 indenfor det samme interval. Ved kørsel af programmet får vi altså en lyd, hvor lydstyrken hæves og sænkes. I blokken tilhørende SOUND QUEUE bruger vi kanal A, envelope-nummer 1, en tone med værdi 200, startlydstyrke på 1 og en varighed på 2 sekunder. Hvis vi ikke havde valgt envelopenummer 1, ville ovenstående brug af SOUND AMPL ENVELOPE ikke have haft effekt på lydstyrken.

På adresse BCBFh ligger rutinen SOUND TONE ENVELOPE,

der afstedkommer ændringer i tonen i stedet for i lydstyrken. Brugen af denne er ækvivalent til brugen af SOUND AMPL ENVELOPE, og vi vil derfor ikke gennemgå den nøjere her. Vi vil i stedet bringe et lille eksempel på rutinen:

Hex-kode	**Mnemonic**
3E01	LD A,1
214F9C	LD HL,40015
CDBFBC	CALL SOUND TONE ENVELOPE
21599C	LD HL,40025
CDAABC	CALL SOUND QUEUE
C9	RET
03	DEFB – ANTAL SEKTIONER –
321E0A	DEFB – 1. SEKTION –
32E20A	DEFB – 2. SEKTION –
0A6432	DEFB – 3. SEKTION –
01	DEFB – KANAL –
00	DEFB – LYDSTYRKEÆNDRING-NUMMER
01	DEFB – TONEÆNDRING-NUMMER –
0000	DEFB – TONE –
00	DEFB – STØJPERIODE –
07	DEFB – LYDSTYRKE –
DC05	DEFB – VARIGHED –

Vi har denne gang valgt at bruge tre sektioner: 50, 30, 10 og 50, – 30, 10 og 10, 100, 50. som du kan se, vil tonen blive dybere og dybere ialt 50 gange, hvorefter den vil blive højere. Sidste sektion bevirker, at tonen bliver dybere igen.

Prøv selv at eksperimentere med både volume envelope og tone envelope på samme tid.

Der findes også andre rutiner i jumpblock'en, som tager sig af lyden. En af disse hedder SOUND RELEASE, og den frigiver en lyd fra en kanal. Når man laver en lyd med SOUND QUEUE, kan man nemlig lægge en lyd på køen, uden at den bliver udført. Hvis bit 6 i kanal status (den første byte i lydblokken), er sat, vil dette være tilfældet. Frigivelsen (eller udførelsen) af denne sker så ved at kalde SOUND RELEASE på BCB3h.

Inden denne kaldes, skal A indeholde bitmønstret for den pågældende kanal (evt. flere). Er bit 0 sat, frigives kanal A, bit 1 frigi-

ver kanal B, mens bit 2 sat frigiver kanal C. Indeholder A-registret værdien 7, vil alle tre kanaler således blive frigivet. Eksempel:

Hex-kode	**Mnemonic**
21539C	LD HL,40019
CDAABC	CALL SOUND QUEUE
CD06BB	CALL KM WAIT CHAR
FE41	CP 65
20F9	JR NZ, – 7
3E01	LD A,1
CDB3BC	CALL SOUND RELEASE
C9	RET
41	DEFB – KANAL –
00	DEFB – LYDSTYRKEÆNDRING-NUMMER
00	DEFB – TONEÆNDRING-NUMMER –
C800	DEFB – TONE –
00	DEFB – STØJPERIODE –
07	DEFB – LYDSTYRKE –
C800	DEFB – VARIGHED –

Find selv ud af, hvilken tast du skal trykke på for at få lyden frigivet. Vi kan hjælpe og sige, at tasten skal være i SHIFT'et tilstand.

Vi bringer et færdigt eksempel på en melodi. Det drejer sig om »Vuffelivov« skrevet af Shu-Bi-Dua. Programmet ser således ud:

Hex-kode	**Mnemonic**
21	DEFB – KANAL –
0000	DEFB – ENVELOPES –
0000	DEFB – TONE –
00	DEFB – STØJPERIODE –
07	DEFB – LYDSTYRKE –
0000	DEFB – VARIGHED –
0C	DEFB – KANAL –
0000	DEFB – ENVELOPES –
0000	DEFB – TONE –
00	DEFB – STØJPERIODE –
07	DEFB – LYDSTYRKE –
0000	DEFB – VARIGHED –

```
21849C          LD HL,40068
064A            LD B,74
C5              PUSH BC
5E              LD E,(HL)
23              INC HL
56              LD D,(HL)
ED53439C        LD (40003),DE
CB3A            SRL D
CB1B            RR E
ED534C9C        LD (40012),DE
23              INC HL
7E              LD A,(HL)
32479C          LD (40007),A
32509C          LD (40016),A
E5              PUSH HL
21409C          LD HL,40000
CDAABC          CALL SOUND QUEUE
30FB            JR NC,-5
21499C          LD HL, 40009
CDAABC          CALL SOUND QUEUE
E1              POP HL
23              INC HL
C1              POP BC
10D4            DJNZ-44
C9              RET
DATA
1C0110FD0010    DEFB - TONER-VARIGHED -
1C0110D50020    DEFB
E10020FD0010    DEFB
1C0120520120    DEFB
1C0130000008    DEFB
1C0120520110    DEFB
1C0120FD0020    DEFB
7B0130000010    DEFB
3F0120520120    DEFB
3F0110520120    DEFB
3F0130520120    DEFB
7B0120AA0110    DEFB
FA0120AA0130    DEFB
000018D50020    DEFB
```

```
E10020FD0012    DEFB
1C0120520120    DEFB
1C0130000008    DEFB
1C0130520110    DEFB
1C0110FD0020    DEFB
7B0130000008    DEFB
3F0120520120    DEFB
3F0110520120    DEFB
3F0130520120    DEFB
7B0110AA0120    DEFB
FA0120AA0130    DEFB
000010AA0110    DEFB
7B0110AA0110    DEFB
5201407B0120    DEFB
AA01107B0140    DEFB
380210520110    DEFB
7B0110AA0150    DEFB
000008AA0110    DEFB
7B0110AA0110    DEFB
5201407B0120    DEFB
AA01107B0130    DEFB
380220520120    DEFB
7B0110AA0150    DEFB
```

Programmet skal kaldes med CALL 40018, idet de 18 første bytes kun er to lydblokke til brug ved SOUND QUEUE-rutinen.

I programmet LOAD'es HL med startadressen for de ialt 74 toner, der skal spilles. Disse er anført bagest i 222 bytes. 1 tone kræver nemlig 3 bytes. De to første angiver tonen, mens den sidste angiver varigheden. Programmet sørger for at hente 3 bytes ind og placere tonen og varigheden på den rette plads i den første lydblok. Dernæst halveres den tone, der er hentet ind, og denne lægges i den anden lydblok. Der opereres således med to lydblokke, der opererer på hver sin kanal, A og C. Bemærk desuden, at lyden til kanal A laver rendez-vous med kanal C og vice versa for at synkronisere tonerne.

Programmet i sig selv burde være enkelt nok at forstå nu, så vi vil overlade det til dig selv at trænge dybere ind og forstå virkningen af de enkelte instruktioner.

Kapitel 18

Vi er nu nået til det mest komplicerede emne indenfor maskinkodeprogrammering, nemlig interrupts. Et interrupt er en afbrydelse af et programs normale gang. Når der genereres et interrupt, bliver det kørende program afbrudt, en såkaldt interruptrutine kaldes, og når denne returnerer, fortsætter hovedprogrammet.

Der er flere forskellige interrupts. Hvis en datamat er tilsluttet en printer, kan den blive styret gennem brug af interrupts. Når printeren er klar til at modtage den næste dataoverførstel, genereres et interrupt, Z80-processoren afbryder det igangværende program, overfører de data, printeren skal bruge, hvorefter der returneres.

Der findes tre forskellige interrupt-tilstande, kaldet »interrupt modes«. Den mest benyttede er interrupt mode 1. Når der i denne tilstand kommer et interrupt, vil adressen på næste instruktion i hovedprogrammet blive lagt på stakken, og der udføres en RST 38h-instruktion. Programmets kørsel fortsætter altså fra adresse 56, indtil instruktionen RETI mødes. Så returneres, den lagrede adresse (returadressen) hentes ned fra stakken, og der fortsættes i hovedprogrammet.

En anden interrupt-tilstand er interrupt mode 0. I denne tilstand skal den enhed, der afbryder, sende en instruktion ud på databussen. Dette betyder, at enheden kan bestemme, hvorfra der skal fortsættes. Sendes f.eks. en RST 10h-instruktion ud på bussen, så vil der fortsættes fra adresse 16. Der returneres stadig, når RETI (RETurn from Interrupt) mødes.

Interrupt mode 2 er den mest avancerede tilstand. Denne tilstand tillader, at flere ydre enheder, som tilsluttes computeren, kan have hver sin interruptrutine. I denne MODE skal den enhed, der afbryder, også udsende en 8 bits-adresse, men denne parres med I-registrets indhold. Det skal forstås på den måde, at I-registret bliver MSB af en 16 bits-adresse, mens enhedens 8 bits bliver LSB. På denne adresse skal da ligge startadressen på den

interruptrutine, som ønskes udført. Dette betyder, at man arbejder med en tabel, som indeholder adresser på alle de mulige interrupt-rutiner. Tabellen skal bare starte på en LIGE adresse. Dette giver en stor fleksibilitet, idet interruptrutinen kan placeres overalt i adresseringsområdet.

Alle disse interrupts hedder BLOKERBARE INTERRUPTS (INT), og det gør de, fordi man kan forhindre dem i at blive udført. Udføres instruktionen DI (Disable Interrupt = frakobl afbrydelse) vil Z80 ikke reagere på det indkommende, blokerbare interrupt, og der fortsættes upåagtet. Med EI (Enable Interrupt = tilkobl afbrydelse) kan man så få Z80 til igen at reagere på et interrupt.

Desuden findes der IKKE-BLOKERBARE INTERRUPT (NMI), og dem er brugeren ikke herre over. Når et sådan genereres, vil der blive udført et kald til adresse 0066h (102). Når interruptrutinen startende på denne adresse returnerer, sker det med RETN i stedet for RETI.

En instruktion ved navn HALT standser programudførelsen. Det, der sker, er, at der udføres en række NOP-instruktioner. HALT-tilstanden kan afbrydes af et reset eller af et interrupt.

Hex-koderne for brugen af interrupt modes m.m. kan ses her:

Hex-kode	**Mnemonic**
ED46	IM 0
ED56	IM 1
ED5E	IM 2
76	HALT
ED4D	RETI
ED45	RETN
F3	DI
FB	EI

Imidlertid kan der genereres interrupt andre steder fra end de ydre enheder. Der kan forekomme et »time interrupt«, d.v.s en afbrydelse af et program regelmæssigt et antal gange i sekundet.

Amstrad CPC-464 har fem sådanne interrupts, men ikke alle er lige velegnede, og vi vil i denne gennemgang kun nævne de vigtige og nyttige.

Det såkaldte »frame flyback interrupt« kommer 50 gange i se-

kundet. Vi vil i slutningen af dette kapitel give et eksempel på brugen af dette interrupt.

Et andet interrupt er det såkaldte »ticker interrupt«, der også kommer 50 gange i sekundet. Keyboardet bruger dette interrupt, idet det scannes for nedtrykkede taster hver gang, ticker interrupt forekommer.

Et tredje interrupt hedder »system clock«, og dette kommer 300 gange i sekundet. I en af de fire jumpblocks er der to rutiner til brug ved dette interrupt. Disse hedder KL TIME SET og KL TIME PLEASE. Man kan med disse rutiner hhv. sætte den indbyggede tæller til en værdi og aflæse værdien. Tælleren forøges 300 gange i sekundet, en gang for hvert interrupt. Du kan selv studere de to rutiner, deres brug og virkemåde i kapitel 21.

Amstrad har imidlertid indbygget flere rutiner til behandling af interrupts. Et interrupt fra hardwaren (maskinellet) er svært at kontrollere, mens et interrupt fra softwaren (programmellet) ville være bedre. Et sådan »software interrupt« kaldes et event. Disse events er beskrevet i blokke, der fortæller noget om typen på det pågældende event, hvilken adresse der skal kaldes (rutinen der kaldes hedder en »event rutine«) m.m. Emnet er meget kompliceret, og vi vil ikke gøre fuldstændig rede for det.

Der findes to typer af events, nemlig de synkrone og de asynkrone. Vi giver, som før nævnt, et eksempel sidst i kapitlet, og det er omhandlende asynkrone events. Vi vil nu gøre rede for de synkrone events, men vi vil ikke give noget færdigt eksempel; blot forklare fremgangsmåden. Forskellen på de to typer er, at asynkrone events ligesom et hardware interrupt udføres præcist, når det bliver genereret. Synkrone events kan udføres, når programmøren finder det belejligt.

Imidlertid er det synkrone event jo genereret, men blot ikke udført. Mens det venter på at blive udført, bliver det lagret i en kø. Samtidig bliver en tæller, den såkaldte »event count«, talt én op, da der er blevet lagt et endnu ikke udført event i køen.

Når et event bliver udført, bliver »event count« derfor decrementeret, så den hele tiden holder styr på antallet af endnu ikke udførte, men genererede events. Der kan dog maksimalt stå 127 events på køen. Når dette tal nås, bliver efterfølgende genererede events ignoreret.

Der findes to forskellige klasser events, nemlig ekspress events

og normale events. Et ekspress event har således højere prioritet end et normal event. Det betyder, at når et ekspress event er genereret og skal lægges i køen, bliver det lagt foran alle normale events, fordi det skal udføres før disse.

Inden for klassen af synkrone events findes en prioritetsrækkefølge. Hvert event har en prioritet mellem 1 og 15, og kommer et event med prioritet 12 f.eks. hen til køen, og der står et event med prioritet 11, så vil det event med prioritet 12 blive lagt foran det med prioritet 11, da det har højere prioritet og derfor skal udføres før.

Eventblokken, som beskriver et events art, indeholder naturligvis også adressen på den eventrutine, som skal udføres, når et event bliver hentet fra køen til udførelse. Denne adresse kan enten være en 3 bytes-adresse eller en 2 bytes-adresse. Det vil normalt være det sidste, og denne ville så angive startadressen på eventrutinen. Den evt. tredje byte skal indeholde en besked, om hvilken ROM der benyttes, men dette vil normalt ikke blive anvendt.

En eventblok er på 7 bytes, men de 3 holder Amstrad selv styr på. De fire andre indeholder en byte om eventtypen, to bytes om eventrutinens startadresse og en sidste byte om ROM. Denne byte vil ved brug af en 2 bytes-adresse blive ignoreret.

Eventblokken initialiseres gennem brug af KL INT EVENT. Når rutinen kaldes, skal

HL indeholde eventblokkens adresse
DE indeholde eventrutinens startadresse
C indeholde nummeret på ROM
B indeholde eventklassen

De enkelte bit i B-registret har denne betydning:

Bit 0 sat betyder, at eventrutinen angives som en 2 bytes-adresse, hvorefter ROM-nummer og derved C-registrets indhold ignoreres (dette bruges ofte)
Bit 1, 2, 3 og 4 angiver prioriteten af det synkrone event (1-15)
Bit 5 skal altid være resat
Bit 6 sat betyder, at der er tale om et ekspress event
Bit 7 sat betyder, at der arbejdes med asynkrone events

Af dette følger, at en resat bit 6 vil give et normalt event, mens en resat bit 7 vil give et synkront event.

Der følger yderligere et par restriktioner på brugen af events. Når der er tale om en 2 bytes-adresse, skal eventrutinen være placeret i de første 48K, så den ikke er lagt »oven i« UROM. Endvidere må en eventrutine ikke bruge det alternative registersæt, som vi omtaler det i det kommende kapitel. IX- og IY-registret må godt bruges, men skal inden returnering med RET-instruktionen have deres startværdier tilbage.

Vi har nu set på, hvordan en eventblok laves, men det er ikke ensbetydende med, at et event er genereret. Det betyder blot, at vi har afgivet besked om, hvordan vort event ser ud.

Et event genereres fra hovedprogrammet ved hjælp af KL EVENT-rutinen. Her skal HL-registret indeholde adressen på eventblokken. Det genererede event vil så blive lagret i køen og vil derfor vente på at blive udført. Det er naturligvis essentielt, at eventblokken er blevet lavet, inden KL EVENT kaldes.

Hovedprogrammet kan undersøge, om der står nogle events på køen, som endnu ikke er udført. Det sker ved at kalde KL POLL SYNCHRONOUS. Hvis der på køen er et event med højere prioritet end det nuværende, vil C-flaget være sat, ellers resat. Hovedprogrammet kan kalde denne rutine, uden det koster meget tid, da rutinen er kort.

Såfremt der står et event, skal det udføres, og da skal kaldes en hel lille løkke:

```
CALL KL NEXT SYNC
RET NC
PUSH HL
PUSH AF
CALL KL DO SYNC
POP AF
POP HL
CALL KL DONE SYNC
JR – 16
```

Denne lille rutine henter et event fra køen, hvis der er noget – ellers returneres der. Så gemmes eventblokkens adresse samt prio-

riteten på det forrige event, eventrutinen udføres, værdierne hentes tilbage, og KL DONE SYNC afslutter eventudførelsen. Dernæst hoppes tilbage for at undersøge, om der er flere events. Disse vil i så fald blive udført.

Vi kan kort opsummere synkrone events på denne måde: først initialiseres en eventblok. I hovedprogrammet kan undersøges, om der står events på køen ved at kalde KL POLL SYNCHRONOUS. Hvis dette er tilfældet, skal vor lille rutine kaldes for at få dem udført. For at få et event op på køen, skal rutinen KL EVENT kaldes.

Som lovet vil vi omtale »frame flyback interrupt«. Vi vil give et eksempel på, hvordan dette interrupt kan bruges til måling af tid.

Der findes en såkaldt »frame flyback queue«. Dette er en kø, hvor et event kan blive placeret. Det vil så blive udført, når et »frame flyback interrupt« kommer, og dette sker som bekendt 50 gange i sekundet. Det bemærkelsesværdige er, at det udførte event ikke flyttes fra køen, når det er udført, men bliver stående klar til at blive udført 1/50 sekund senere. Det pågældende event kan dog blive flyttet af en rutine i en jumpblock, men det skal altså gøres af brugeren.

Med rutinen KL NEW FRAME FLY skabes et event på samme måde, som vi allerede har set det, men denne rutine nøjes ikke med at initialisere eventblokken. Den sætter det samtidig på listen, i køen, over events, som skal udføres hver gang, der kommer et »frame flyback interrupt«. Igen skal HL LOAD'es med eventblokkens startadresse, DE med eventrutinens startadresse, og B med eventtypen (C med ROM-nummeret). Eventblokken, som herved opstår, er på 9 bytes, hvoraf de 5 ikke benyttes af brugeren. De 4 andre er præcis mage til dem, som vi omtalte tidligere i dette kapitel under eventblokproblematikken.

At det på listen stående event udføres 50 gange i sekundet skyldes, at vi her har beskrevet asynkrone events (faktisk normale asynkrone events, idet ekspress asynkrone events ikke er brugervenlige). Det betyder, at det på listen stående event automatisk udføres, lige før Z80 forlader interruptrutinen. Det skal således ikke aktiveres af programmøren, som vi så det ved et synkron event. Det gælder desuden, at et indkommende interrupt under udførelsen af en eventrutine vil blive udført efter eventrutinens afslutning, før der returneres til hovedprogrammet.

Man kan således benytte et »frame flyback interrupt« som en tidmåler, idet det 50 gange i sekundet vil kalde en eventrutine. Vi vil nu give et lille, simpelt eksempel på dette.

Programmet ser således ud:

Hex-kode	**Mnemonic**
213F9C	LD HL,39999
3632	LD (HL),50
21659C	LD HL,40037
11519C	LD DE,40017
0681	LD B,129
CDD7BC	CALL KL NEW FRAME FLY
C9	RET
213F9C	LD HL,39999
35	DEC (HL)
C0	RET NZ
3632	LD (HL),50
3E07	LD A,7
CD5ABB	CALL TXT OUTPUT
C9	RET
21659C	LD HL,40037
CDDDBC	CALL KL DEL FRAME FLY
C9	RET

Programmet skal ligge fra adresse 40000.

Vi vil nu gennemgå programmet i detaljer, så du kan blive helt fortrolig med opbygningen. Den egentlige rutine, som bruges, er:

```
LD HL,40037
LD DE,40017
LD B,129
CALL KL NEW FRAME FLY
RET
```

Denne rutine gør, at et event sættes på listen over de, der skal udføres ved et »frame flyback interrupt«. HL LOAD'es med 40037, og det er startadressen på eventblokken. De 9 bytes, som eventblokken fylder, skal ligge et sted, hvor den ikke bliver forstyrret,

og vi har lagt den fra adresse 40037, idet den sidste byte i vort program ligger på 40036.

DE er LOAD'et med adressen på eventrutinen, og den ligger i 40017; adressen efter hovedprogrammets ophør. BC LOAD'es med 129. Derved opnås et asynkront event med 2 bytes-adresse, hvorved C's værdi bliver ligegyldig. Dernæst kaldes KL NEW FRAME FLY, der initialiserer blokken og lægger det pågældende event på køen.

Eventrutinen starter på 40017, og den ser således ud:

```
LD HL,39999
DEC (HL)
RET NZ
LD (HL),50
LD A,7
CALL TXT OUTPUT
RET
```

Denne rutine kaldes altså 50 gange i sekundet. På adresse 39999 ligger en tæller, der holder øje med, hvor mange gange der er kaldt. Hver gang der kaldes, decrementeres den, og bliver den ikke nul, returneres der øjeblikkeligt. Tællerens startværdi er sat i starten af programmet med

```
LD HL,39999
LD (HL),50
```

og den er blevet sat til 50. Derfor vil eventrutinen, de første 49 gange den kaldes, returnere ved RET NZ, da tælleren ikke er blevet decrementeret fra 50 til 0.

Når dette sker, er der gået et sekund siden det sidste gang skete, og tælleren skal følgelig atter have startværdien 50. Samtidig udføres en PRINT CHR $ (7), og det betyder, at den indbyggede højttaler giver en lille lyd fra sig. Dernæst returneres der.

Prøv nu at køre programmet, hvis du da ikke allerede har gjort det.

For hvert sekund kommer der altså en lille lyd fra højttaleren, men kan det ikke stoppes? Jo, naturligvis:

```
LD HL,40037
CALL KL DEL FRAME FLY
RET
```

Denne rutine fjerner vort event fra listen. HL indeholder, som du kan se, startadressen på eventblokken.

Når du er blevet træt af lyden og ikke er tilfreds med blot at skrue ned for den, så kan du fjerne vort event ved at kalde rutinen på 40030 – dér ligger nemlig ovenstående rutine.

Bemærk i øvrigt, at det er muligt at arbejde helt normalt med Amstrad, selv om den skal kalde eventrutinen 50 gange i sekundet.

Det er naturligvis også muligt at have flere events liggende på listen, f.eks. ét, der udføres hvert sekund, ét, der udføres hvert femte etc.

Vi vil senere bruge ovenstående teknik til et ur. Det kan du læse mere om i kapitel 22.

Til sidst vil vi lige kort fortælle om rutinen på adresse 3Bh. Den er ligesom RST 30h tilgængelig for brugeren, men den har med interrupts at gøre. Hvis du har nogle eksterne enheder, som genererer interrupts, vil computeren automatisk kalde denne adresse (3Bh). Det er op til brugeren at have en rutine liggende, som behandler dette interrupt. LROM bliver frakoblet, inden der kaldes. Yderligere interrupts er også frakoblet (disabled), og det skal de forblive med, til rutinen er færdig. Til sidst kan vi sige, at med disse events er det muligt at have f.eks. 4 farver i skærm-MODE 2 på en gang, 8 i MODE 1 etc.

En anden mulighed er, at BORDER'en kan få *flere forskellige* farver. Det trick man bruger, er at afbryde skærmbilledrutinens arbejde, tildele INK'ene nogle andre værdier og lade skærmbilledrutinen fortsætte. Skærmen kan på denne måde blive delt i 2 vandrette dele, hvor f.eks. INK 0 er sort i øverste del, men rød i nederste.

Alt dette kræver dog, at man sender farverne direkte ud gennem portene og det er ikke det allernemmeste at forstå. Derfor vil vi ikke liste et sådant program, men overlade det til dig selv, når du har fået mere (*meget* mere) erfaring.

Kapitel 19

I dette kapitel behandler vi de alternative registre, som du første gang stiftede bekendtskab med i oversigten over registre i kapitel 3.

Der findes 8 alternative registre i Z80-processoren, og de har næsten samme navne som nogle af de registre, vi allerede har arbejdet med. Navnene på de alternative registre er A', F', B', C', D', E', H' og L'. Læg mærke til det lille mærke oppe i hjørnet. Dette indikerer, at det er et alternativt register. De fungerer på præcis samme måde som de almindelige. De kan således sættes sammen til 16 bits-registre o.s.v.

Man kan skifte mellem det alternative og det normale registersæt med instruktionerne

Hex-kode	**Mnemonic**
08	EX AF,AF'
D9	EXX

Med EX AF, AF' ombyttes A med A' og F med F'. EXX-instruktionen foretager flere ombytninger, idet både B, C, D, E, H og L erstattes af de tilsvarende alternative. Når disse to instruktioner er udført, vil de normale registre blive gemt væk, mens de nye registre virker på præcis samme måde som de gamle. For at få de normale tilbage, bruger man igen de 2 instruktioner. Bemærk i øvrigt, at man ikke behøver at bruge begge instruktioner på én gang. Man kan altså godt nøjes med kun at bruge A' og F' med EX AF,AF'. Desuden skal det nævnes, at når registrene hentes frem igen, vil de beholde den værdi, som de havde, da de blev gemt væk. At benytte det alternative registersæt vil således normalt give to hold registre, som man så kan skifte imellem, idet registrene beholder værdien, når de gemmes.

Når vi skriver, at man »normalt« har to sæt, så betyder det, at Amstrad afviger lidt herfra. Den bruger nemlig selv de alternati-

ve registre til nogle af interruptrutinerne i operativsystemet, som vi har beskrevet så grundigt. Den er bestemt ikke glad for, at brugeren roder i disse registre eller ihvertfald ikke i visse af dem. Operativsystemet bruger nemlig kun BC' og F' til at gemme nogle *meget* vigtige tal i. De øvrige registre bruges til at lave mellemresultater i. B', C' og F' må derfor for enhver pris ikke ændres, ellers sker der forfærdelige ting og sager!

I B' ligger nogle I/O-adresser, mens C' indeholder nogle oplysninger vedrørende ROM's status (tilkobling/frakobling) og skærmens MODE. De enkelte bit har betydning på denne måde:

Bit 0-1: Skærmens tilstand (MODE)
Bit 2 : Sat medfører frakobling af LROM
Bit 3 : Sat medfører frakobling af UROM
Bit 4-7: Har betydning for portene

I F-registret er det kun C-flaget, som betyder noget. Det er normalt resat. Når det er sat, indikerer det, at en interrupt-rutine er ved at blive udført. Selv om der er utilfredshed fra ROM's side, kan det godt lade sig gøre at bruge de alternative registre. Du kan f.eks. bruge denne lille sekvens:

```
DI
EXX
EX AF,AF'
PUSH AF
PUSH BC
...
opgave
...
POP BC
POP AF
EX AF,AF'
EXX
EI
```

Der bruges en DI-instruktion i starten, fordi det ville have været katastrofalt, hvis et interrupt blev accepteret, når registrene B', C' og F' ikke har deres »rigtige« værdi. Selvfølgelig PUSH'er vi for

senere at kunne hente værdierne tilbage. Sidst i sekvensen tillades interrupts igen.

Der er alligevel ulemper ved metoden, for når et interrupt ikke tillades, vil mange ting ikke virke mere. Dette gælder for eksempel tastaturscanningen og de blinkende farver, da begge disse anordninger benytter sig af interrupts. Derfor må rutinen, som vi lægger inden i vor sekvens fra før, være ret kort rent tidsmæssigt. Den anden ulempe er, at der ikke må kaldes rutiner i operativsystemet (firmwarerutiner). Dette betyder, at du ikke så nemt og smertefrit kan få noget ud på skærmen, PEN kan ikke ændres o.s.v.

Så vi er nødt til at bruge en anden teknik, hvis vi skal bruge interrupts. Inden fremgangsmåden vises, skal vi lige repetere vor viden om interrupts. Når et sådant indtræffer, vil rutinen på adresse 56 (38h) kaldes. På adresse 56 står der en JP-instruktion. Hvorhen den hopper, behøver du ikke bekymre dig om! Vi skal nemlig lave om på det! Inden vi går i gang, skal vi have reserveret 12 bytes til at gemme nogle registre i samt til at henlægge den føromtalte adresse i JP-instruktionen på adresse 56. Det betyder intet, hvor disse 12 bytes ligger, så vi har valgt at bruge nedenstående, symbolske adresser:

Adresse: Bruger AF' – 2 bytes til at gemme brugerens AF'
Adresse: Bruger BC' – 2 bytes til at gemme brugerens BC'
Adresse: Bruger DE' – 2 bytes til at gemme brugerens DE'
Adresse: Bruger HL' – 2 bytes til at gemme brugerens HL'
Adresse: Firmw BC' – 2 bytes til at gemme firmware's BC'
Adresse: Firm-Int – 2 bytes til at gemme den rigtige interruptrutineadresse

Først skal nedenstående rutine, som vi har kaldt BRUGER, indtastes:

```
DI
EXX
EX AF,AF'
LD (Firmw BC'), BC
LD HL,(57)
LD (Firm-Int),HL
```

```
LD HL, Bruger-Interrupt
LD (57),HL

LD HL,(Bruger AF')
PUSH HL
POP AF
LD BC,(Bruger BC')
LD DE,(Bruger DE')
LD HL,(Bruger HL')
EX AF,AF'
EXX
EI
RET
```

Denne rutine gemmer først operativsystemets BC'-register samt adressen på dens interrupt-rutine. I stedet for denne lægger vi en ny adresse ned (Bruger-Interrupt), så når der indtræffer et interrupt, vil vores rutine kaldes. Vi skal nok vise, hvad denne rutine skal indeholde. Til sidst henter vi vores egne alternative registerværdier frem. Igen husker vi, at interrupts bliver ignoreret, mens disse instruktioner udføres.

Vi skal også bruge nedenstående rutine. Vi har kaldt den FIRMWARE:

```
DI
EXX
EX AF,AF'
LD (Bruger BC'),BC
LD (Bruger DE'),DE
LD (Bruger HL'),HL
PUSH AF
POP HL
LD (Bruger AF'),HL
LD HL, (Firm-Int)
LD (57),HL
LD BC,(Firmw BC')
AND A
EX AF,AF'
```

```
EXX
RET
```

Vi starter med at gemme vores alternative registre, og derefter lægger vi den normale interruptrutines adresse tilbage. Derefter hentes operativsystemets BC' tilbage, og carry-flaget resættes med AND A. Til sidst returneres. Læg mærke til, at vi ikke medtager en EI igen. Interrupts må nemlig ikke accepteres endnu!

I din interrupt-rutine skal du indsætte dette:

```
CALL FIRMWARE
CALL 38h
CALL BRUGER
RET
```

Den adresse, du lagrer i 57 og 58, skal være startadressen på ovenstående instruktioner. Når interrupt indtræffer, vil disse instruktioner altså blive kaldt. Først sørges for at få Amstrads eget BC'-register tilbage og gemme dine egne registres værdier. Desuden lægges adressen på den originale interruptrutine tilbage. Interruptrutinen kaldes så med CALL 38h, og til sidst gemmes Amstrads BC', og vores værdier hentes tilbage. Der sørges endvidere for, at det er vores interruptrutine, der kaldes næste gang, der kommer et interrupt. Til sidst returneres med RET, så den normale programudførelse fortsætter.

Hvis man i sit program ønsker at bruge det alternative registersæt, udføres

```
CALL BRUGER
```

og når de er brugt, benyttes

```
CALL FIRMWARE
EI
```

Interrupts tillades igen.

Inden du prøver at udføre LD (57),HL, skal du sikre dig, at LROM er frakoblet (disabled). Ellers vil der ikke ske noget, da

man jo ikke kan berøre ROM's indhold. HL lægges jo på 57 og 58 i RAM!

Denne metode kunne altså være anvendelig, da interrupts tillades. Imidlertid kan ROM-kald ikke accepteres. Dette vil i længden være utilfredsstillende, så vi kan ændre lidt på rutinerne fra før, så ROM-kald bliver muligt. Det er præcis den samme fremgangsmåde til at bruge de alternative registre, men når en ROM-rutine skal kaldes, må dette bruges:

```
CALL FIRMWARE
EI
CALL ROM-rutine
CALL BRUGER
```

Der er indsat en EI-instruktion, fordi interrupt i rutinen FIRMWARE blev frakoblet, og dette skulle ændres. Som du kan se, kræver det 10 bytes hver gang en ROM-rutine skal kaldes, og det er lidt trættende i længden. Er der rigtig mange kald vil det tillige være pladskrævende. Det kan dog også kortes ned til 5 bytes, hvis denne metode bruges:

```
CALL SPAR 5
adresse
```

hvor adressen, der skal kaldes, ligger på de to bytes lige efter CALL-instruktionen. SPAR 5-rutinen ser således ud:

```
CALL FIRMWARE
EXX
POP HL
LD E,(HL)
INC HL
LD D,(HL)
INC HL
PUSH HL
LD HL,BRUGER
PUSH HL
PUSH DE
```

```
EXX
EI
RET
```

Først sørges for, at firmware's BC' tilbagekaldes. Derefter skifter vi over til de alternative registre (dog ikke AF'). Vi kan nu bruge DE og HL, da kun BC bruges af Amstrad. Husk i øvrigt, at interrupts ikke er tilladt. Derefter hentes returadressen ned. På denne adresse står jo den adresse, som vi egentlig ville kalde. Denne adresse hentes ned i DE, og adressen efter de to bytes PUSH'es (dertil skal der returneres på et tidspunkt). HL LOAD'es med startadressen på BRUGER, og den PUSH'es. DE indeholdende ROM-rutine-adressen PUSH'es også. Der skiftes registersæt, og interrupts tillades. Til sidst står en RET, og den vil tage værdien øverst på stakken og fortsætte programudførelsen derfra – adressen er ROM-rutinens startadresse. Når ROM-rutinen møder en RET, tager den den næste værdi fra stakken, som er BRUGER's startadresse. Når den rutine afsluttes med en RET findes næste returadresse, som er den første værdi vi PUSH'ede. Det er adressen efter de to bytes, som angav ROM-rutinen. Det hele er nu, som det skal være, men bemærk den kraftige manipulation af returadresser.

Som du forhåbentlig har indset, er det en indviklet affære at bruge de alternative registre på Amstrad, men der er absolut ingen tvang for dig. Det er helt frivilligt at bruge de alternative registre. Du kan – evt. når du er blevet skrappere – eksperimentere videre med det grundlag, du har fået, men vær forsigtig!

Kapitel 20

Selv erfarne programmører kan ikke eliminere risikoen for fejl, men de kan gøre den mindre! I dette kapitel vil vi først se på, hvordan man kan prøve at undgå fejl og så derefter finde dem, der opstår.

Når et maskinkodeprogram indeholder en fejl, så er der to muligheder. Enten så *crasher* computeren, og det bliver nødvendigt at starte påny, eller resultatet bliver fejlagtigt. Den sidste type fejl er absolut den mest behagelige. En fejl kan normalt skyldes to ting. Det mest normale er, at der er begået en fejl i selve rutinen. Der mangler en instruktion, programmøren har glemt en detalje ved et register, eller et flag har ikke den rigtige status. Disse fejl er de almindeligste og de rareste. Den anden type er den, hvor selve ideen bag rutinen ikke virker. Når en maskinkodeprogrammør sætter sig for at lave et program, så har denne altid en idé til programmets opbygning. Hvis denne idé ikke virker, så vil det medføre fejl, naturligvis, men desuden kræver det, at hele programmet tages op til ny revidering. Ofte med brug af papirkurven til følge!

Heldigvis vil den fundamentale idé bag programmet ofte være rigtig, og derved vil en evt. fejl normalt ikke kunne ændre ved programmets basis. Idealet er selvfølgelig helt at undgå at lave fejl. Dette er kort sagt umuligt at give en færdig opskrift på! Imidlertid vil brug af de råd, som vi nu giver, hjælpe med til at fjerne risikoen for fejl.

Først og fremmest skal du undgå at lave indviklede programmer. Lad i starten være med at tænke på at spare bytes. Det første mål må være at kontrollere, at din idé til programmet er korrekt, og derefter kan du så forbedre det. Et større program vil altid bestå af flere små rutiner, der tilsammen udgør det samlede program. Sørg for i starten, at du har mulighed for at afprøve disse rutiner én efter én. Lav dem som underprogrammer, som du kan kalde fra en hovedrutine, når du begynder at få samlet stum-

perne. Herved vil en fejl være lettere at finde, da området, hvori den kan ligge, formindskes. Afprøvning af små rutiner er altid at foretrække. Hvis du benytter hex-loaderen som indtastningsmiddel, så giv dig god plads. Læg ikke nødvendigvis rutinerne i en lang række. Hvis du pludselig mangler instruktioner i én af dem, så skal det hele ændres. Herved skal du også ændre f.eks. CALL-adresser. Lad være med det – det er bare en kilde til fejl. Læg rutinerne rundt omkring i hukommelsen med god plads imellem, så der er mulighed for at indsætte glemte instruktioner.

Prøv desuden at undgå JP-instruktioner. JR-instruktionerne vil være at foretrække, da du herved kan flytte rundt på programmet. Det bliver relokerbart. I den sidste ende, når de afprøvede dele af programmet skal samles til et hele, vil det være en fordel at undgå at ændre adresser. Det kan blive nødvendigt ved CALL-instruktioner men prøv at minimere brugen af instruktioner, der kræver disse ændringer. Hovedprogrammet, der kalder alle de små rutiner, vil naturligvis bestå af CALL-instruktioner, men ofte vil dette så også være det eneste sted, hvor rettelser er påkrævede.

Desuden skal du huske altid at lægge dine rutiner over det sted, hvor BASIC-programmer kan operere. MEMORY-kommandoen sikrer dette. Herved bliver du ikke forstyrret i dine rutiner. Nu har vi omtalt de råd, som vi finder mest nærliggende at videregive med henblik på elimination af fejl under programmeringen. Imidlertid kan du hjælpe dig selv ved at nedskrive kommentarer til dine rutiner. Hvis du skal vende tilbage til et program flere dage senere, vil det være rart at have programmets idé og virkemåde nedskrevet, så du hurtigt atter kan sætte dig ind i rutinen. I disse kommentarer kan også indgå, hvilket formål de forskellige registre har i programmet. Dette vil lette dig i den sidste ende, og det kræver ikke stort arbejde.

Vi har allerede nævnt, at du skal afprøve alle de små underprogrammer én efter én. Du kan indtaste dem efterhånden, som du producerer dem. Inden afprøvningen er der en ting, der er helt afgørende, at du husker: programmet skal gemmes på bånd eller diskette. Dette er så essentielt, at vi ikke kan understrege det nok. Hvis der er en fejl i programmet, er risikoen for et crash stort, og da vil det være af uvurderlig betydning, at du har en kopi af det crashede program. Dette kan nu atter indlæses i com-

puteren – husk MEMORY-kommandoen – og du står klar til at finde fejlen.

Hvad er den hyppigste fejl? Det er vel nok en stakfejl. Som du ved er det vigtigt at holde 100% kontrol med stakken, da returadressen til BASIC er placeret deroppe. Hvis der sker en fejl her, vil et crash næsten være givet på forhånd. Derfor skal du, når en fejl er opstået, starte med at tage en udskrift af programmet og tælle stakinstruktioner (PUSH og POP). Hvis du ikke benytter flere stakke eller manipulerer med den, så skal der være lige mange PUSH- og POP-instruktioner. Dette er en hyppig fejl.

Hvis det ikke er stakken, der er skyld i fejlen, så prøv at kontrollere alle JR og DJNZ. Her er det vigtigt, at den afstand, hvormed der hoppes, er helt præcis. Kort sagt: kontroller alle relative hop. Hvis dette ikke afhjælper fejlen, så må du teste flagenes status. Ofte vil du i et program benytte et betinget hop eller kald, og her skal du sikre dig, at du netop skal udføre hoppet (eller kaldet) på den skrevne betingelse og ikke på den modsatte. En fjerde kilde til fejl er, at ROM-rutinerne spolerer registrene. Kapitel 21 indeholder en fuldstændig oversigt over, hvilke registre der ødelægges ved de enkelte rutiner. Stol i første omgang ikke på, at ROM-rutinen bevarer dine registres værdi. Måske bruger den dem selv. Kig ihvertfald omhyggeligt efter i kapitel 21 en gang til.

Nu har vi givet dig 4 meget hyppige fejlkilder, og hvis ingen af dem har kunnet spore din fejl, så begynder det at blive sværere. På nuværende tidspunkt vil vi råde dig til at kontrollere hex-koderne. Der er flere muligheder. Måske har du slået en forkert hex-kode op (eller din hukommelse var ikke tilstrækkelig god?) eller du har simpelt hen indtastet en hex-kode for lidt. Vi vil med andre ord på dette tidspunkt i fejlfindingen råde dig til at kontrollere alle de indtastede hex-koder samt undersøge, om en i programmet er forkert. Hvis du arbejder med en disassembler, kan du kontrollere de anvendte mnemonics direkte. Vi vil ikke i denne bog anvise dig, hvordan du kan lave en Z80-dissassembler.

Hvis dette ikke hjælper dig til at finde fejlen, så må du til at kontrollere selve programmet fra en ende af. Her kan det være til hjælp at lave en såkaldt skrivebordskørsel. Skriv på et stykke papir alle registrene op sammen med stakken, og start fra en ende af. Tag instruktionerne en efter en, ændr i registrenes ind-

hold, hvis instruktionerne gør det. Metoden er besværlig, men den vil ofte føre til fejlen. Din kørsel er altså simpelt hen en manuel kørsel af programmet, hvor du undervejs kan holde øje med de forskellige registre. I kapitel 24 bringer vi et komplet program som hjælp til en sådan skrivebordskørsel. Det program, kaldet SINGLESTEP, vil tage en instruktion ad gangen, og på skærmen kan du så holde øje med registrene og stakkens indhold. Mere om det i kapitel 24.

Hvis dette ikke har afhjulpet din fejl endnu, så må du dele underrutinerne endnu finere op ved at indsætte RET-instruktioner undervejs (husk at holde øje med stakken). Så kan du skridt for skridt næsten prøve endnu en programstump. Området, hvori fejlen kan være, konkretiseres derfor.

Tilbage står kun, hvis du stadig ikke er hjulpet (det vil du være i praktisk talt alle tilfælde), at hente naboen, sætte ham ind i situationen, og lade ham teste dit program.

Hvis du derimod undervejs finder fejlen, så ret det med det samme. Tag et hurtigt blik over situationen (kan der være flere fejl af samme slags?) og tag en ny kopi. Afprøv programmet påny. Er der stadig fejl, så må du forfra i fejlfindingen.

På et tidspunkt – jo mere rutineret du er, desto tidligere – har du fundet alle fejl (hvis der var nogen!), og nu står kun det triste arbejde tilbage. En seriøs programmør vil altid på dette sted renskrive programmet, opstille det fra en ende af, så det fylder mindst muligt i hukommelsen, for derefter at lagre det på bånd. Så vil vedkommende sætte sig tilbage i stolen og glæde sig over succesen for sejrsikker at eksekvere sit færdige produkt – for at konstatere, at det ikke virker!!! Du skal altid huske, at den sidste renskrivning kan indebære fejl. Det vil ofte være adressefejl (ved CALL f.eks.), der hurtigt kan rettes. Husk dog: vær ikke sikker på programmets succes før den endelige afprøvning er foretaget.

Hertil kommer, at perfektionisten vil fortsætte med at tænke over måder til forkortelse af programmet, så det kan fylde mindre eller arbejde hurtigere. Lad ham da bare gøre det, hvis han har lyst . . .

Vi bringer en fuldstændig gennemgang af alle instruktioner i Z80-instruktionssættet. Vi forklarer deres virkemåde, men bringer ingen mnemonics. Dem må du finde i appendiks A. Brug

denne del af kapitlet som opslagsværk til de forskellige instruktioners formål og funktion.

ADC: Findes i både 8 og 16 bits-form. I begge tilfælde lægges to værdier sammen. Resultatet tillægges én, hvis C-flaget er sat. Ved 8 bit adderes på A, mens HL benyttes ved 16 bits-addition.

ADD: Findes i både 8 og 16 bits-form. I begge tilfælde lægges to værdier sammen. Ved 8 bit adderes på A, mens HL, IX eller IY bruges ved 16 bits-addition.

AND: De enkelte bit i A sammenlignes med en værdi i et register, på en adresse eller blot et givet tal. Resultatet af hver bitsammenligning er 1, hvis begge de sammenlignede bit er 1. Resultatet lægges i A.

BIT: En bit i et register testes. Z-flaget antager den modsatte værdi af den testede bit.

CALL: En underrutine kaldes, og der vendes tilbage med RET. Instruktionen kan gøres betinget af 4 flags status. Det drejer sig om C-, Z-, P/V- og S-flaget.

CCF: C-flagets værdi vendes. Hvis det er sat, resættes det, og vice versa.

CP: A sammenlignes med et register eller et tal. Ingen registre påvirkes. Er de lig hinanden, sættes Z-flaget, og C-flaget resættes. Er A størst, resættes begge. Er A mindst, sættes C-flaget, mens Z-flaget resættes.

CPD: Virker som CP (HL), men efterfulgt af DEC HL og DEC BC. P/V-flaget resættes, hvis BC bliver nul. Z-flaget resættes, hvis A er lig (HL).

CPDR: Virker som CPD, men der repeteres enten til BC er nul eller til værdien i A findes i den gennemsøgte blok af bytes. Flagene påvirkes på samme måde som ved CPD.

CPI: Virker som CPD, men HL incrementeres.

CPIR: Virker som CPDR, men HL incrementeres.

CPL: Hver enkelt bit i A vendes.

DAA: Gør et resultat efter addition, subtraktion m.m. gyldigt i BCD-aritmetik.

DEC: Værdien i et register (8 eller 16 bits) gøres én mindre.

DI: Forhindrer Z80 i at reagere på interrupts.

DJNZ: B-registret decrementeres. Hvis B ikke bliver nul, hoppes et angivet antal bytes mellem – 128 og 127 frem (tilbage).

EI: Tillader Z80 at reagere på interrupts.

EX: Ombytter toppen af stakken med HL, IX eller IY, eller ombytter DE med HL. Endelig kan AF ombyttes med AF'.

EXX: Ombytter de alternative registre BC', DE', og HL' med de fungerende BC, DE og HL.

HALT: Stopper programudførelsen indtil et reset eller et interrupt.

IM: Angiver interrupt-tilstand (0, 1 eller 2).

IN: Tager indholdet på en givet port ind i A eller indholdet på en port specificeret i C ind i et 8 bits-register.

IND: Udfører IN (HL),(C), hvor (C) er indholdet på den port, som register C peger på, efterfulgt af DEC B og DEC HL.

INDR: Virker som IND, men der repeteres, til B er nul.

INI: Virker som IND, men HL incrementeres.

INIR: Virker som INDR, men HL incrementeres.

INC: Værdien i et register (8 eller 16 bits) forøges med én.

JP: Springer til en adresse mellem 0 og 65535, der er angivet. Kan også springe til adressen i HL, IX eller IY. Instruktionen kan gøres betinget af C, Z, P/V og S's status.

JR: Hopper et angivet antal bytes mellem – 128 og 127 frem (tilbage). Kan gøres betinget af flagene C og Z.

LD: Bruges til at give registre (alle på nær F-registret) værdi. En værdi kan også lægges på en adresse, eller værdien på en adresse kan lægges i et register.

LDD: Udfører LD (DE),(HL) efterfulgt af DEC HL, DEC DE og DEC BC.

LDDR: Virker som LDD, men repeterer til BC er nul.

LDI: Udfører det samme som LDD, men DE og HL incrementeres.

LDIR: Virker som LDDR, men DE og HL incrementeres.

NEG: Hver enkelt bit i A vendes, og der tillægges én.

NOP: Udfører ingenting.

OR: De enkelte bit i A sammenlignes med en værdi i et register, på en adresse eller blot et givet tal. Resultatet af hver bitsammenligning er 1, hvis blot den ene af de sammenlignede bit er 1. Resultatet lægges i A.

OUT: Indholdet i A lægges ud på en givet port, eller indholdet fra et 8 bits-register lægges på den port, som C-registret peger på.

OUTD: Udfører OUT (C),(HL) efterfulgt af DEC HL og DEC B.

OTDR: Virker som OUTD, men repeterer til B er nul.

OUTI: Virker som OUTD, men HL incrementeres.

OTIR: Virker som OTDR, men HL incrementeres.

POP: Henter værdien øverst på stakken ned i et 16 bits-register.

PUSH: Lægger værdien i et 16 bits-register øverst på stakken.

RES: Resætter en bit i et register.

RET: Bruges til at returnere fra en underrutine. Der kan vendes tilbage til BASIC med denne instruktion, der kan gøres betinget af fire flags status, nemlig C, Z, P/V og S.

RETI: Der returneres fra en interruptrutine (INT).

RETN: Der returneres fra en interruptrutine (NMI).

RL: Hver bit i et 8 bits-register flyttes én til venstre. Bit 7 flyttes til C-flaget og bit 0 tager C-flagets værdi.

RLA: Virker som RL A.

RLC: Hver bit i et 8 bits-register flyttes én til venstre. Bit 7 flyttes både til C-flaget og til bit 0.

RLCA: Virker som RLC A.

RLD: Flytter de fire første bit i A til de fire første i (HL), der flyttes til de fire sidste bit i (HL), der flyttes til de fire første i A.

RR: Hver bit i et 8 bits-register flyttes én til højre. Bit 0 går til C-flaget, og C-flagets værdi til bit 7.

RRA: Virker som RR A.

RRC: Hver bit i et 8 bits-register flyttes én til højre. Bit 0 går både til C-flaget og til bit 7.

RRCA: Virker som RRC A.

RRD: De fire første bit i A flyttes til de fire sidste (bit 4-7) i (HL), der flyttes til de fire første i (HL), der flyttes til de fire første i A.

RST: Kalder forskellige rutiner, hvis startadresser er fastlagt af Zilog.

SBC: Der findes både 8 og 16 bits-form. Ved 8 bits-subtraktion subtraheres på A, mens HL benyttes ved 16 bits. Der trækkes yderligere én fra, hvis C-flaget er sat.

SET: Sætter en bit i et register.

SLA: Hver enkelt bit i et 8 bits-register flyttes til venstre. Bit 7 går til C-flaget, og bit 0 resættes.

SRA: Hver enkelt bit i et 8 bits-register flyttes til højre. Bit 0 går til C-flaget, og bit 7 berøres ikke.

SRL: Hver enkelt bit i et 8 bits-register flyttes til højre. Bit 0 går til C-flaget, og bit 7 resættes.

SUB: Findes kun i 8 bits-form. Der subtraheres på A-registret.

XOR: De enkelte bit i A sammenlignes med en værdi i et register, på en adresse eller blot et givet tal. Resultatet af hver bitsammenligning er 1, hvis én og kun én af de sammenlignede bits er 1. Resultatet lægges i A.

Oversigten er kun summarisk. En fuldstændig forklaring på de enkelte instruktioners virkemåde kan du finde i vor gennemgang tidligere i bogen.

Kapitel 21

Dette kapitel omhandler de rutiner, der ligger i de såkaldte jumpblocks. Du kan benytte det som et opslagsværk til at finde de forskellige ROM-rutiners karakteristika og særheder.

Der findes fire forskellige jumpblocks, og de hedder således:

1) »The Main Jumpblock« – indeholder 190 rutiner, der er beliggende fra BB00h til BD39h.
2) »Firmware Indirections« – indeholder 13 rutiner, der er beliggende fra BDCDh til BDF3h.
3) »The High Kernel Jumpblock« – indeholder 12 rutiner, der er beliggende fra B900h til B923h.
4) »The Low Kernel Jumpblock« – indeholder 16 rutiner, der er beliggende fra 0000h til 003Bh.

Vi vil i dette kapitel kun gøre rede for rutinerne i »The Main Jumpblock« og i »The High Kernel Jumpblock«. I førstnævnte ligger, som det ses ovenfor, langt de fleste af rutinerne. Dér er placeret rutinerne til behandling af tastatur, tekst- og grafikskærm, lyd og kassettebåndoptager m.m. En brøkdel af disse har vi gjort rede for i kapitlerne 14-17. Dette er således en komplet redegørelse for samtlige rutiners virkemåde. I »The High Kernel Jumpblock« ligger bl.a. rutiner, der til- og frakobler LROM og UROM. Vi har ikke medtaget uddybende kommentarer om de to øvrige jumpblocks. I »Firmware Indirections« ligger rutiner, som rutinerne i den primære jumpblock benytter. Disse behøver brugeren ikke have kendskab til. Rutinerne i den fjerde jumpblock, »The Low Kernel Jumpblock«, har vi allerede omtalt i kapitel 13, idet det er bl.a. RST-rutinerne. En fuldstændig gennemgang af disse er således givet.

Vi bringer i dette kapitel først en oversigt over rutinerne. Her bringes rutinernes navne og deres numre. Disse numre kan så bruges som udgangspunkt til at finde den enkelte rutine i den dy-

bere gennemgang, der følger. Desuden står adressen angivet.

Under selve rutinerne angives også navn, adresse og nummer, og desuden forklares dens formål. Der beskrives, hvilke værdier registrene eventuelt skal have, inden rutinen kaldes, samt hvilke registre, der ødelægges af rutinen, og som derfor ikke bevarer sin værdi fra før kaldet. Nederst angives evt. uddybende kommentarer. Vi gennemgår her et eksempel:

KM GET REPEAT (21) – BB3Ch – 47932
Undersøger, om en tast er repeterende
FØR KALD:
A skal indeholde tastnummeret
VED RETURNERING:
Z-flaget resat, hvis tasten er repeterende
Z-flaget sat, hvis tasten ikke er repeterende
C-flaget resat
A, HL og øvrige flag ødelægges
Bemærk: Hvis tastnummeret er ugyldig (større end 79), vil Z-flagets tilstand være meningsløs.

Øverst angives altså navnet på rutinen, og den hedder KM GET REPEAT. Den er nummer 21 i sin gruppe, og den er beliggende i adresse 47932, der er BB3C i hexadecimal notation. Dens formål ses derunder. Det er at afgøre, hvorvidt en tast er repeterende eller ej. Inden rutinen kaldes, skal A-registret indeholde nummeret på den tast, der skal undersøges. Når rutinen returnerer, vil Z-flaget afgøre, hvorvidt den er repeterende eller ej. Endvidere ses det, at C-flaget altid er resat. A, øvrige flag og HL berøres af rutinen, hvorved deres værdi fra før kaldet ikke er bevaret. Det gælder derimod ikke indholdet i registrene BC, DE, IX og IY, idet disse ikke er nævnt. Under bemærkningerne nævnes, at rutinen accepterer et meningsløst tastnummer og returnerer med et meningsløst resultat.

Således angives altså rutinerne. Som tidligere sagt: brug dette kapitel som opslagsværk til rutinerne, og vær opmærksom på de angivelser, der er ved hver enkelt rutine.

OVERSIGT OVER RUTINERNE I »THE MAIN JUMPBLOCK«:

Gruppe 1: Tastaturet

1:	BB00	– KM INITIALISE
2:	BB03	– KM RESET
3:	BB06	– KM WAIT CHAR
4:	BB09	– KM READ CHAR
5:	BB0C	– KM CHAR RETURN
6:	BB0F	– KM SET EXPAND
7:	BB12	– KM GET EXPAND
8:	BB15	– KM EXP BUFFER
9:	BB18	– KM WAIT KEY
10:	BB1B	– KM READ KEY
11:	BB1E	– KM TEST KEY
12:	BB21	– KM GET STATE
13:	BB24	– KM GET JOYSTICK
14:	BB27	– KM SET TRANSLATE
15:	BB2A	– KM GET TRANSLATE
16:	BB2D	– KM SET SHIFT
17:	BB30	– KM GET SHIFT
18:	BB33	– KM SET CONTROL
19:	BB36	– KM GET CONTROL
20:	BB39	– KM SET REPEAT
21:	BB3C	– KM GET REPEAT
22:	BB3F	– KM SET DELAY
23:	BB42	– KM GET DELAY
24:	BB45	– KM ARM BREAK
25:	BB48	– KM DISARM BREAK
26:	BB4B	– KM BREAK EVENT

Gruppe 2: Tekstskærmen

1:	BB4E	– TXT INITIALISE
2:	BB51	– TXT RESET
3:	BB54	– TXT VDU ENABLE
4:	BB57	– TXT VDU DISABLE
5:	BB5A	– TXT OUTPUT

6: BB5D – TXT WR CHAR
7: BB60 – TXT RD CHAR
8: BB63 – TXT SET GRAPHIC
9: BB66 – TXT WIN ENABLE
10: BB69 – TXT GET WINDOW
11: BB6C – TXT CLEAR WINDOW
12: BB6F – TXT SET COLUMN
13: BB72 – TXT SET ROW
14: BB75 – TXT SET CURSOR
15: BB78 – TXT GET CURSOR
16: BB7B – TXT CUR ENABLE
17: BB7E – TXT CUR DISABLE
18: BB81 – TXT CUR ON
19: BB84 – TXT CUR OFF
20: BB87 – TXT VALIDATE
21: BB8A – TXT PLACE CURSOR
22: BB8D – TXT REMOVE CURSOR
23: BB90 – TXT SET PEN
24: BB93 – TXT GET PEN
25: BB96 – TXT SET PAPER
26: BB99 – TXT GET PAPER
27: BB9C – TXT INVERSE
28: BB9F – TXT SET BACK
29: BBA2 – TXT GET BACK
30: BBA5 – TXT GET MATRIX
31: BBA8 – TXT SET MATRIX
32: BBAB – TXT SET M TABLE
33: BBAE – TXT GET M TABLE
34: BBB1 – TXT GET CONTROLS
35: BBB4 – TXT STR SELECT
36: BBB7 – TXT SWAP STREAMS

Gruppe 3: Grafikskærmen

1: BBBA – GRA INITIALISE
2: BBBD – GRA RESET
3: BBC0 – GRA MOVE ABSOLUTE
4: BBC3 – GRA MOVE RELATIVE
5: BBC6 – GRA ASK CURSOR

6: BBC9 – GRA SET ORIGIN
7: BBCC – GRA GET ORIGIN
8: BBCF – GRA WIN WIDTH
9: BBD2 – GRA WIN HEIGHT
10: BBD5 – GRA GET W WIDTH
11: BBD8 – GRA GET W HEIGHT
12: BBDB – GRA CLEAR WINDOW
13: BBDE – GRA SET PEN
14: BBE1 – GRA GET PEN
15: BBE4 – GRA SET PAPER
16: BBE7 – GRA GET PAPER
17: BBEA – GRA PLOT ABSOLUTE
18: BBED – GRA PLOT RELATIVE
19: BBF0 – GRA TEST ABSOLUTE
20: BBF3 – GRA TEST RELATIVE
21: BBF6 – GRA LINE ABSOLUTE
22: BBF9 – GRA LINE RELATIVE
23: BBFC – GRA WR CHAR

Gruppe 4: Skærmen

1: BBFF – SCR INITIALISE
2: BC02 – SCR RESET
3: BC05 – SCR SET OFFSET
4: BC08 – SCR SET BASE
5: BC0B – SCR GET LOCATION
6: BC0E – SCR SET MODE
7: BC11 – SCR GET MODE
8: BC14 – SCR CLEAR
9: BC17 – SCR CHAR LIMITS
10: BC1A – SCR CHAR POSITION
11: BC1D – SCR DOT POSITION
12: BC20 – SCR NEXT BYTE
13: BC23 – SCR PREV BYTE
14: BC26 – SCR NEXT LINE
15: BC29 – SCR PREV LINE
16: BC2C – SCR INK ENCODE
17: BC2F – SCR INK DECODE
18: BC32 – SCR SET INK

19: BC35 – SCR GET INK
20: BC38 – SCR SET BORDER
21: BC3B – SCR GET BORDER
22: BC3E – SCR SET FLASHING
23: BC41 – SCR GET FLASHING
24: BC44 – SCR FILL BOX
25: BC47 – SCR FLOOD BOX
26: BC4A – SCR CHAR INVERT
27: BC4D – SCR HW ROLL
28: BC50 – SCR SW ROLL
29: BC53 – SCR UNPACK
30: BC56 – SCR REPACK
31: BC59 – SCR ACCESS
32: BC5C – SCR PIXELS
33: BC5F – SCR HORIZONTAL
34: BC62 – SCR VERTICAL

Gruppe 5: Kassettebåndoptageren

1: BC65 – CAS INITIALISE
2: BC68 – CAS SET SPEED
3: BC6B – CAS NOISY
4: BC6E – CAS START MOTOR
5: BC71 – CAS STOP MOTOR
6: BC74 – CAS RESTORE MOTOR
7: BC77 – CAS IN OPEN
8: BC7A – CAS IN CLOSE
9: BC7D – CAS IN ABANDON
10: BC80 – CAS IN CHAR
11: BC83 – CAS IN DIRECT
12: BC86 – CAS RETURN
13: BC89 – CAS TEST EOF
14: BC8C – CAS OUT OPEN
15: BC8F – CAS OUT CLOSE
16: BC92 – CAS OUT ABANDON
17: BC95 – CAS OUT CHAR
18: BC98 – CAS OUT DIRECT
19: BC9B – CAS CATALOG
20: BC9E – CAS WRITE

21: BCA1 – CAS READ
22: BCA4 – CAS CHECK

Gruppe 6: Lyden

1: BCA7 – SOUND RESET
2: BCAA – SOUND QUEUE
3: BCAD – SOUND CHECK
4: BCB0 – SOUND ARM EVENT
5: BCB3 – SOUND RELEASE
6: BCB6 – SOUND HOLD
7: BCB9 – SOUND CONTINUE
8: BCBC – SOUND AMPL ENVELOPE
9: BCBF – SOUND TONE ENVELOPE
10: BCC2 – SOUND A ADDRESS
11: BCC5 – SOUND T ADDRESS

Gruppe 7: The Kernel

1: BCC8 – KL CHOKE OFF
2: BCCB – KL ROM WALK
3: BCCE – KL INIT BACK
4: BCD1 – KL LOG EXT
5: BCD4 – KL FIND COMMAND
6: BCD7 – KL NEW FRAME FLY
7: BCDA – KL ADD FRAME FLY
8: BCDD – KL DEL FRAME FLY
9: BCE0 – KL NEW FAST TICKER
10: BCE3 – KL ADD FAST TICKER
11: BCE6 – KL DEL FAST TICKER
12: BCE9 – KL ADD TICKER
13: BCEC – KL DEL TICKER
14: BCEF – KL INIT EVENT
15: BCF2 – KL EVENT
16: BCF5 – KL SYNC RESET
17: BCF8 – KL DEL SYNCHRONOUS
18: BCFB – KL NEXT SYNC
19: BCFE – KL DO SYNC
20: BD01 – KL DONE SYNC

21: BD04 – KL EVENT DISABLE
22: BD07 – KL EVENT ENABLE
23: BD0A – KL DISARM EVENT
24: BD0D – KL TIME PLEASE
25: BD10 – KL TIME SET

Gruppe 8: Machine Pack

1: BD13 – MC BOOT PROGRAM
2: BD16 – MC START PROGRAM
3: BD19 – MC WAIT FLYBACK
4: BD1C – MC SET MODE
5: BD1F – MC SCREEN OFFSET
6: BD22 – MC CLEAR INKS
7: BD25 – MC SET INKS
8: BD28 – MC RESET PRINTER
9: BD2B – MC PRINT CHAR
10: BD2E – MC BUSY PRINTER
11: BD31 – MC SEND PRINTER
12: BD34 – MC SOUND REGISTER

Gruppe 9:

1: BD37 – JUMP RESTORE

OVERSIGT OVER RUTINERNE I »THE HIGH KERNEL JUMPBLOCK«:

1: B900 – KL U ROM ENABLE
2: B903 – KL U ROM DISABLE
3: B906 – KL L ROM ENABLE
4: B909 – KL L ROM DISABLE
5: B90C – KL ROM RESTORE
6: B90F – KL ROM SELECT
7: B912 – KL CURR SELECTION
8: B915 – KL PROBE ROM
9: B918 – KL ROM DESELECT
10: B91B – KL LDIR

11: B91E – KL LDDR
12: B921 – KL POLL SYNCHRONOUS

OVERSIGT OVER RUTINERNE I »THE LOW KERNEL JUMPBLOCK«:

0000: RESET ENTRY (RST 00h)
0008: LOW JUMP (RST 08h)
000B: KL LOW PCHL
000E: PCBC INSTRUCTION
0010: SIDE CALL (RST 10h)
0013: KL SIDE PCHL
0016: PCDE INSTRUCTION
0018: FAR CALL (RST 18h)
001B: KL FAR PCHL
001E: PCHL INSTRUCTION
0020: RAM LAM (RST 20h)
0023: KL FAR ICALL
0028: FIRM JUMP (RST 28h)
0030: USER RESTART (RST 30h)
0038: INTERRUPT ENTRY (RST 38h)
003B: EXT INTERRUPT

»THE MAIN JUMPBLOCK«

GRUPPE 1: TASTATURET

KM INITIALISE (1) – BB00h – 47872
Initialiserer tastaturet som ved start af computeren
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges.
Bemærk: Initialiseringen omfatter bl.a. retablering af karakter-

koder og udvidelseskarakterer. Keybufferen tømmes, og tasternes status med hensyn til repetition og repetitionshastighed sættes til deres starttilstand.

KM RESET (2) – BB03h – 47875
Resætter key bufferen m.m.
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Rutinen sletter udvidelseskaraktererne, og keybufferen tømmes. Desuden frakobles f.eks. muligheden for at afbryde med ESC-tasten.

KM WAIT CHAR (3) – BB06h – 47878
Venter på den næste karakter fra tastaturet
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat
A indeholder karakteren
Andre flag ødelægges
Bemærk: Rutinen venter på den næste karakter enten fra keybufferen eller fra en udvidelseskarakter. Med sidstnævnte vil rutinen returnere med en enkelt karakter for hver gang den kaldes.

KM READ CHAR (4) – BB09h – 47881
Tester, om en karakter er tilgængelig fra tastaturet
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis en karakter er tilgængelig
  A indeholder karakteren
C-flaget resat, hvis en karakter ikke var tilgængelig
  A ødelægges
Andre flag ødelægges
Bemærk: Rutinen tester, om der er en karakter i keybufferen eller fra en udvidelseskarakter. Denne rutine venter ikke på karakteren, men returnerer øjeblikkelig.

KM CHAR RETURN (5) – BB0Ch – 47884
Gemmer en karakter til kald af KM WAIT CHAR eller KM READ CHAR
FØR KALD:
A skal indeholde karakteren, der skal gemmes
VED RETURNERING:
Alt bevares
Bemærk: Karakteren i A vil blive returneret ved kald af KM WAIT CHAR eller KM READ CHAR før alle andre karakterer. Det er kun muligt at have en karakter af denne slags (en såkaldt »put back«-karakter). Kald af rutinen to gange i træk vil medføre, at den første karakter går tabt. Karakteren med koden 255 kan ikke gemmes, da denne betyder »ingen 'put back'-karakter«.

KM SET EXPAND (6) – BB0Fh – 47887
Definerer en udvidelseskarakter
FØR KALD:
B skal indeholde nummeret på udvidelseskarakteren
C skal indeholde længden af udvidelseskarakteren
HL skal indeholde startadressen for dens udseende
VED RETURNERING:
C-flaget sat, hvis udvidelsen er godkendt
C-flaget resat, hvis udvidelsen ikke er godkendt
A, BC, DE, HL og øvrige flag ødelægges
Bemærk: Udvidelsen kan være ugyldig, hvis den er for lang, eller hvis der ikke er tale om en udvidelseskarakter overhovedet. B skal således ligge indenfor intervallet 128-159.

KM GET EXPAND (7) – BB12h – 47890
Tester for en karakter i en udvidelseskarakter
FØR KALD:
A skal indeholde nummeret på udvidelseskarakteren
L skal indeholde nummeret på den plads i udvidelseskarakteren, der skal undersøges
VED RETURNERING:
C-flaget sat, hvis karakteren er fundet
  A indeholder karakteren
C-flaget resat, hvis karakteren ikke er fundet
  A ødelægges

DE og andre flag ødelægges
Bemærk: Den første plads i udvidelseskarakteren har nummer 0. C-flaget er resat, enten hvis karakteren ikke blev fundet, eller hvis nummeret på udvidelseskarakteren er ugyldigt (det skal ligge i intervallet 128-159).

KM EXP BUFFER (8) – BB15h – 47893
Opretter en buffer for udvidelseskarakterer
FØR KALD:
DE skal indeholde adressen for bufferen
HL skal indeholde længden af bufferen
VED RETURNERING:
C-flaget sat, hvis bufferens oprettelse er godkendt
C-flaget resat, hvis bufferen er for kort
A, BC, DE, HL og øvrige flag ødelægges
Bemærk: Rutinen opretter en buffer til udvidelseskaraktererne. Denne må ikke ligge under ROM, og den skal mindst være 49 bytes lang. Hvis den er for kort i forhold til udvidelseskaraktererne, resættes C-flaget, og den gamle expansion buffer fortsætter.

KM WAIT KEY (9) – BB18h – 47896
Venter på den næste tast fra tastaturet
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat
A indeholder karakteren eller udvidelsestegnet
Øvrige flag ødelægges
Bemærk: Rutinen returnerer med den næste karakter i key bufferen. Er dette koden for en udvidelseskarakter, returneres med udvidelsestegnet (det tegn, som koden normalt henfører til). Udvidelseskarakterer accepteres altså ikke. Rutinen returnerer først, når en karakter er fundet.

KM READ KEY (10) – BB1Bh – 47899
Tester, om en karakter er tilgængelig fra tastaturet
FØR KALD:
Ingen krav

VED RETURNERING:
C-flaget sat, hvis en karakter var tilgængelig
A indeholder karakteren eller udvidelsestegnet
C-flaget resat, hvis ingen karakter var tilgængelig
A ødelægges
Andre flag ødelægges
Bemærk: Denne rutine returnerer øjeblikkelig, hvad enten en karakter er i keybufferen eller ej. For at få en forklaring på udvidelseskarakter/udvidelsestegn henvises til KM WAIT KEY (rutine 9 i denne gruppe).

KM TEST KEY (11) – BB1Eh – 47902
Tester, om en tast er nedtrykket
FØR KALD:
A skal indeholde tastnummeret
VED RETURNERING:
Z-flaget resat, hvis tasten var nedtrykket
Z-flaget sat, hvis den ikke var nedtrykket
C-flaget resat
C indeholder SHIFT- og CONTROL-status
A, HL og øvrige flag ødelægges
Bemærk: Rutinen undersøger, hvorvidt en given tast eller given del af joystick er nedtrykket. C-registret returnerer med status for SHIFT og CONTROL. Hvis bit 7 er sat, er CONTROL nedtrykket, mens bit 5 er sat, hvis SHIFT er nedtrykket. Tastnummeret undersøges ikke. Er dette ugyldigt, returneres med den korrekte status for SHIFT og CONTROL, men tastens status er meningsløs.

KM GET STATE (12) – BB21h – 47905
Aflæser status for CAPS LOCK og SHIFT LOCK
FØR KALD:
Ingen krav
VED RETURNERING:
L indeholder SHIFT LOCK-status
H indeholder CAPS LOCK-status
AF ødelægges
Bemærk: SHIFT LOCK betyder, at alle taster behandles, som var SHIFT nedtrykket, mens CAPS LOCK betyder, at alle alfabetiske karakterer skrives med stort. Deres status returneres således:

0 betyder »ikke låst«
255 betyder »låst«.

KM GET JOYSTICK (13) – BB24h – 47908
Aflæser status af tilsluttede joystick(s)
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder status af joystick 0
H indeholder status af joystick 0 (H = A)
L indeholder status af joystick 1
Flagene ødelægges
Bemærk: Status af joysticks returneres således:

bit 0 sat: op
bit 1 sat: ned
bit 2 sat: venstre
bit 3 sat: højre
bit 4 sat: knap 2 nedtrykket
bit 5 sat: knap 1 nedtrykket
bit 6 sat: reserveknap
bit 7 : altid resat

Joystick 1's tastnumre kan ikke skelnes fra visse af tastaturets tasters numre.

KM SET TRANSLATE (14) – BB27h – 47911
Sætter hvilken karakter en tast skal henføre til, når hverken SHIFT eller CONTROL er nedtrykket
FØR KALD:
A skal indeholde tastnummeret
B skal indeholde den nye karakterkode
VED RETURNERING:
AF og HL ødelægges
Bemærk: Hvis tastnummeret er større end 79, er det ulovligt, og der sker intet. Med karakterkoder inden for følgende intervaller skal der bemærkes:

128-159: Dette er en udvidelseskarakter. Kald af KM READ CHAR eller KM WAIT CHAR og tryk på tasten vil medføre den til koden hørende udvidelseskarakter. Kald af KM READ KEY eller KM WAIT KEY vil medføre den normale karakter (udvidelsestegnet).

240-252: Disse har betydning for BASIC ved editering, behandling af markør og stop (breaks).

253: Dette er koden for CAPS LOCK, og den medfører, at CAPS LOCK skifter status.

254: Dette er koden for SHIFT LOCK, og den medfører, at denne skifter status.

255: Denne kode betyder, at tasten ignoreres.

KM GET TRANSLATE (15) – BB2Ah – 47914

Undersøger, hvilken karakter en given tast henfører til, når hverken SHIFT eller CONTROL er nedtrykket

FØR KALD:

A skal indeholde tastnummeret

VED RETURNERING:

A indeholder den tilhørende karakterkode

HL og flagene ødelægges

Bemærk: Se rutinen KM SET TRANSLATE.

KM SET SHIFT (16) – BB2Dh – 47917

Sætter hvilken karakter en tast skal henføre til, når SHIFT er nedtrykket (eller SHIFT LOCK er »låst«) og CONTROL ikke er nedtrykket

FØR KALD:

A skal indeholde tastnummeret

B skal indeholde karakterkoden

VED RETURNERING:

AF og HL ødelægges

Bemærk: Se rutinen KM SET TRANSLATE.

KM GET SHIFT (17) – BB30h – 47920

Undersøger, hvilken karakter en given tast henfører til, når SHIFT er nedtrykket (eller SHIFT LOCK er »låst«) og CONTROL ikke er nedtrykket

FØR KALD:
A skal indeholde tastnummeret
VED RETURNERING:
A indeholder den tilhørende karakterkode
HL og flagene ødelægges
Bemærk: Se KM SET TRANSLATE.

KM SET CONTROL (18) – BB33h – 47923
Sætter hvilken karakter en tast skal henføre til, når CONTROL er nedtrykket
FØR KALD:
A skal indeholde tastnummeret
B skal indeholde karakterkoden
VED RETURNERING:
AF og HL ødelægges
Bemærk: Se KM SET TRANSLATE.

KM GET CONTROL (19) – BB36h – 47926
Undersøger, hvilken karakter en given tast henfører til, når CONTROL er nedtrykket
FØR KALD:
A skal indeholde tastnummeret
VED RETURNERING:
A indeholder den tilhørende karakterkode
HL og flagene ødelægges
Bemærk: Se KM SET TRANSLATE.

KM SET REPEAT (20) – BB39h – 47929
Bestemmer, hvorvidt en tast er repeterende
FØR KALD:
B skal indeholde 255, hvis tasten skal være repeterende
B skal indeholde 0, hvis tasten ikke skal være repeterende
A skal indeholde tastnummeret
VED RETURNERING:
AF, BC og HL ødelægges
Bemærk: Hvis tastnummeret er ulovligt, større end 79, sker der intet.

KM GET REPEAT (21) – BB3Ch – 47932
Undersøger, om en tast er repeterende
FØR KALD:
A skal indeholde tastnummeret
VED RETURNERING:
Z-flaget resat, hvis tasten er repeterende
Z-flaget sat, hvis tasten ikke er repeterende
C-flaget resat
A, HL og øvrige flag ødelægges
Bemærk: Hvis tastnummeret er ugyldigt (større end 79), vil Z-flagets tilstand være meningsløs.

KM SET DELAY (22) – BB3Fh – 47935
Sætter repetitionshastigheden samt repetitionsforsinkelsen (den tid, der går, inden tasten repeterer)
FØR KALD:
H skal indeholde repetitionsforsinkelsen
L skal indeholde repetitionshastigheden
VED RETURNERING:
AF ødelægges
Bemærk: Tallene skal angives i antal hele scanninger af tastaturet. Der foregår 50 sådanne scanninger i sekundet. Ved start er repetitionsforsinkelsen 30 (0,6 sek.), mens repetitionshastigheden er 2 (0,04 sek. = 25 karakterer pr. sekund). En angivelse på 0 betragtes som 256. Repetitionshastigheden og -forsinkelsen berører alle repeterende taster.

KM GET DELAY (23) – BB42h – 47938
Undersøger repetitionshastighed og repetitionsforsinkelse
FØR KALD:
Ingen krav
VED RETURNERING:
H indeholder repetitionsforsinkelsen
L indeholder repetitionshastigheden
AF ødelægges
Bemærk: Tallene angives i antal scanninger af tastaturet. Disse forekommer 50 gange i sekundet. Se i øvrigt rutinen KM SET DELAY (rutine 22 i denne gruppe).

KM ARM BREAKS (24) – BB45h – 47941
Tillader stop (breaks)
FØR KALD:
DE skal indeholde adressen på »break event«-rutinen
C skal indeholde ROM-nummer
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Rutinen tillader generering af breaks, der kan forekomme f.eks. ved tryk på ESC-tasten.

KM DISARM BREAK (25) – BB48h – 47944
Forhindrer stop (breaks)
FØR KALD:
Ingen krav
VED RETURNERING:
AF og HL ødelægges
Bemærk: Rutinen forhindrer generering af breaks. Disse kan igen tillades ved brug af KM ARM BREAKS-rutinen.

KM BREAK EVENT (26) – BB4Bh – 47947
Genererer et stop (break), hvis det er tilladt
FØR KALD:
Ingen krav
VED RETURNERING:
AF og HL ødelægges
Bemærk: Denne rutine forsøger at udføre et break event. Hvis dette er tilladt, genereres et break event som angivet i KM ARM BREAKS. Hvis det ikke er tilladt (breaks er forhindret f.eks. ved KM DISARM BREAKS), sker der intet.

GRUPPE 2: TEKSTSKÆRMEN

TXT INITIALISE (1) – BB4Eh – 47950
Total initialisering af tekstskærmen som ved start af computeren
FØR KALD:
Ingen krav

VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Initialiseringen er fuldstændig og omfatter således bl.a. følgende: baggrunden sættes til ink 0, pennen til ink 1, stream 0 vælges, vinduet fylder hele skærmen, alle brugerdefinerbare karakterer slettes og markøren placeres i øverste venstre hjørne.

TXT RESET (2) – BB51h – 47953
Resætter dele af tekstskærmen
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Der resættes kun dele, deriblandt kontrolkoderne, der sættes til deres startværdier. Pen, baggrund, vinduer, streams m.m. berøres ikke.

TXT VDU ENABLE (3) – BB54h – 47956
Tillader skrivning af karakterer på tekstskærmen
FØR KALD:
Ingen krav
VED RETURNERING:
AF ødelægges
Bemærk: Rutinen tillader udskrivning af karakterer. Tilladelsen henfører til den nuværende stream. Desuden tillades markørens tilstedeværelse på skærmen.

TXT VDU DISABLE (4) – BB57h – 47959
Forhindrer udskrivning af karakterer på tekstskærmen
FØR KALD:
Ingen krav
VED RETURNERING:
AF ødelægges
Bemærk: Rutinen forhindrer udskrivning af karakterer. Den berører kun den benyttede stream. Markørens tilstedeværelse på skærmen forhindres ved at kalde TXT CUR DISABLE.

TXT OUTPUT (5) – BB5Ah – 47962
Udsender en karakter (til skærmen) eller en kontrolkode

FØR KALD:
A skal indeholde den ønskede karakter
VED RETURNERING:
Alt bevares
Bemærk: Hvis karakterkoden er mellem 0 og 31, er der tale om en kontrolkode, og den udføres som en sådan. Disse kan have op til 9 parametre. De udsendte koder gemmes så i en buffer (kontrolkode buffer), indtil der er det korrekte antal parametre. Hvis tekstskærmen er frakoblet (ved kald af TXT VDU DISABLE), udskrives intet, men kontrolkoder udføres. Karaktererne udskrives i den nuværende stream.

TXT WR CHAR (6) – BB5Dh – 47965
Skriver en karakter på skærmen
FØR KALD:
A skal indeholde den karakter, der skal udskrives
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Hvis karakterkoden er mellem 0 og 31 udskrives den tilhørende karakter. Den behandles således ikke som en kontrolkode. Hvis tekstskærmen er frakoblet, vil karakteren ikke blive udskrevet. Efter udskriften flyttes markørens position en til højre.

TXT RD CHAR (7) – BB60h – 47968
Aflæser karakteren på den nuværende cursorposition i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis rutinen fandt en genkendelig karakter
  A indeholder karakterkoden
C-flaget resat, hvis der ingen genkendelig karakter var
  A indeholder 0
Øvrige flag ødelægges
Bemærk: Markørens position tvinges ikke indenfor et vindue, inden karakteren aflæses. Matricen for den læste karakter sammenlignes med matricerne for karaktererne i karaktersættet. Hvis skærmens aflæste karakter ikke findes i karaktersættet, er

den uigenkendelig. Dette kan bl.a. skyldes ændring i baggrund eller grafik, f.eks. en streg gennem karakteren.

TXT SET GRAPHIC (8) – BB63h – 47971
Til- eller frakobler muligheden for udskrivning af grafiske karakterer i den nuværende stream
FØR KALD:
A skal være forskellig fra 0, hvis grafikudskrivning skal tilkobles.
A skal indeholde 0, hvis grafikudskrivning skal frakobles
VED RETURNERING:
AF ødelægges
Bemærk: Når udskrivning af grafiske karakterer er tilladt, vil karakterer til TXT OUTPUT blive udskrevet via grafikskærmen i stedet for via tekstskærmen. Kontrolkoder bliver derfor udskrevet og ikke adlydt. Hvis grafikudskrivning er tilkoblet, vil kald af TXT VDU DISABLE ikke medføre, at udskrivning forhindres.

TXT WIN ENABLE (9) – BB66h – 47974
Sætter størrelsen på vinduet i den nuværende stream
FØR KALD:
H skal indeholde det ene kolonnenummer
D skal indeholde det andet kolonnenummer
L skal indeholde det ene rækkenummer
E skal indeholde det andet rækkenummer
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Kolonne- og rækkenumrene skal angives i forhold til skærmens øverste, venstre hjørne, der har koordinaterne (0,0). Den venstre kant af vinduet bliver den mindste værdi af H og D, mens den øverste kant bliver den mindste af L og E. Markøren flyttes til vinduets øverste, venstre hjørne, men det clear'es ikke. Ved start fylder vinduet hele skærmen.

TXT GET WINDOW (10) – BB69h – 47977
Undersøger størrelsen af vinduet i den nuværende stream samt om det dækker hele skærmen
FØR KALD:
Ingen krav

VED RETURNERING:
C-flaget resat, hvis vinduet dækker hele skærmen
C-flaget sat, hvis vinduet fylder mindre end hele skærmen
H indeholder kolonnenummeret for den venstre kant
D indeholder kolonnenummeret for den højre kant
L indeholder rækkenummeret for den øverste kant
E indeholder rækkenummeret for den nederste kant
A ødelægges
Bemærk: Koordinaterne gives i forhold til skærmens øverste, venstre hjørne, der har koordinaterne (0,0).

TXT CLEAR WINDOW (11) – BB6Ch – 47980
Clear'er vinduet i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Vinduet clear'es til den nuværende streams baggrundsfarve. Markøren flyttes til vinduets øverste venstre hjørne.

TXT SET COLUMN (12) – BB6Fh – 47983
Sætter markørens vandrette position
FØR KALD:
A skal indeholde den ønskede kolonne
VED RETURNERING:
AF og HL ødelægges
Bemærk: Kolonnenummeret for markøren i den nuværende stream sættes. Kolonnenummeret skal gives i logiske koordinater, hvormed menes, at kolonnen længst til venstre har kolonnenummer 1. Hvis markøren placeres uden for vinduet i den pågældende stream, vil den blive tvunget indenfor, inden en karakter udskrives.

TXT SET ROW (13) – BB72h – 47986
Sætter markørens lodrette position
FØR KALD:
A skal indeholde den ønskede række
VED RETURNERING:
AF og HL ødelægges

Bemærk: Rækkenummeret skal gives i logiske koordinater. Se TXT SET COLUMN for yderligere forklaring.

TXT SET CURSOR (14) – BB75h – 47989
Sætter markørens position i den nuværende stream
FØR KALD:
H skal indeholde det ønskede kolonnenummer
L skal indeholde det ønskede rækkenummer
VED RETURNERING:
AF og HL ødelægges
Bemærk: Markørens position skal angives i logiske koordinater. Se TXT SET COLUMN for yderligere anvisninger.

TXT GET CURSOR (15) – BB78h – 47992
Undersøger markørens position i den nuværende stream samt det antal gange vinduet har rullet
FØR KALD:
Ingen krav
VED RETURNERING:
H indeholder kolonnenummeret
L indeholder rækkenummeret
A indeholder værdien af en tæller til brug ved scroll
Flagene ødelægges
Bemærk: Markørens position er angivet i logiske koordinater, se evt. TXT SET COLUMN for yderligere forklaring. Tælleren i A (roll-count) er blevet decrementeret hver gang, vinduet er rullet op, og incrementeret hver gang, vinduet er rullet ned. Markørpositionen, der angives, behøver nødvendigvis ikke være det sted, hvor der næste gang udskrives, da denne kan ligge uden for vinduet, og derfor vil blive tvunget indenfor ved udskrivningen.

TXT CUR ENABLE (16) – BB7Bh – 47995
Tillader markørens tilstedeværelse på skærmen i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
AF ødelægges
Bemærk: Denne rutine er til brug for programmøren.

TXT CUR DISABLE (17) – BB7Eh – 47998
Forhindrer markørens tilstedeværelse på skærmen i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
AF ødelægges
Bemærk: Denne rutine er til brug for programmøren.

TXT CUR ON (18) – BB81h – 48001
Tillader markørens tilstedeværelse på skærmen i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
Alt bevares
Bemærk: Denne rutine er til brug for ROM.

TXT CUR OFF (19) – BB84h – 48004
Forhindrer markørens tilstedeværelse på skærmen i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
Alt bevares
Bemærk: Denne rutine er til brug for ROM.

TXT VALIDATE (20) – BB87h – 48007
Undersøger, om en markørposition er inden for vinduet
FØR KALD:
H skal indeholde kolonnenummeret
L skal indeholde rækkenummeret
VED RETURNERING:
C-flaget sat, hvis udskrivning på positionen ikke vil medføre scroll
B ødelægges
C-flaget resat, hvis udskrivning på positionen vil medføre scroll
B indeholder 255 for scroll opad
B indeholder 0 for scroll nedad

H indeholder kolonnenummeret på positionen, hvor karakteren vil blive udskrevet
L indeholder rækkenummeret på positionen, hvor karakteren vil blive udskrevet
A og øvrige flag ødelægges
Bemærk: Markørens position skal angives i logiske koordinater, d.v.s. hvor øverste, venstre hjørne har (1,1). Hvis markørpositionen ligger uden for vinduet i den nuværende stream, bestemmes hvor der næste gang vil blive udskrevet efter at positionen er blevet tvunget inden for vinduet.

TXT PLACE CURSOR (21) – BB8Ah – 48010
Placerer en markør på markørpositionen på den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
AF ødelægges
Bemærk: Denne rutine tillader således placering af flere markører i et vindue. Hvis markørpositionen er uden for vinduet, tvinges den indenfor, inden markøren udskrives.

TXT REMOVE CURSOR (22) – BB8Dh – 48013
Fjerner en markør på markørpositionen i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
AF ødelægges
Bemærk: Denne rutine må kun fjerne markører sat af TXT PLACE CURSOR. Markøren skal fjernes, når markørens position ændres.

TXT SET PEN (23) – BB90h – 48016
Sætter tekst pennen i den nuværende stream
FØR KALD:
A skal indeholde den ønskede pen
VED RETURNERING:
AF og HL ødelægges

Bemærk: Den ønskede pen gøres legal, hvis den ligger uden for det tilladte område. I mode 0 foretages en AND 15-instruktion, i mode 1 en AND 3, og i mode 2 en AND 1-instruktion.

TXT GET PEN (24) – BB93h – 48019
Undersøger tekst pennen i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder den benyttede pen
Flagene ødelægges.

TXT SET PAPER (25) – BB96h – 48022
Sætter tekst baggrunden i den nuværende stream
FØR KALD:
A indeholder den ønskede baggrund
VED RETURNERING:
AF og HL ødelægges
Bemærk: Den ønskede baggrund legaliseres, se evt. rutinen TXT SET PEN. Ved clearing af vinduet i denne stream vil blive benyttet denne baggrundsfarve. Denne rutine medfører ikke i sig selv denne clearing.

TXT GET PAPER (26) – BB99h – 48025
Undersøger tekst baggrunden i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder den benyttede baggrund
Flagene ødelægges

TXT INVERSE (27) – BB9Ch – 48028
Ombytter pen og baggrund i den nuværende stream
FØR KALD:
Ingen krav
VED RETURNERING:
AF og HL ødelægges
Bemærk: Markøren tegnes ikke igen efter kald af denne rutine, hvorfor den ikke bør være på skærmen, når rutinen kaldes.

TXT SET BACK (28) – BB9Fh – 48031
Vælger udskrivningstilstand
FØR KALD:
A skal være 0 for uigennemsigtig tilstand
A skal være forskellig fra 0 for gennemsigtig tilstand
VED RETURNERING:
AF og HL ødelægges
Bemærk: Ved uigennemsigtig tilstand skrives karakteren på den til stream'en hørende baggrund. Ved gennemsigtig tilstand skrives direkte oven på det, der i forvejen står på skærmen. Se i øvrigt kontrolkode 22.

TXT GET BACK (29) – BBA2h – 48034
Undersøger udskrivningstilstanden
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder 0, hvis udskrivningstilstand er uigennemsigtig
A indeholder en værdi forskellig fra 0, hvis tilstanden er gennemsigtig
DE, HL og flagene ødelægges
Bemærk: Tilstanden påvirker kun den nuværende stream. Se i øvrigt TXT SET BACK og kontrolkode 22 for yderligere anvisninger.

TXT GET MATRIX (30) – BBA5h – 48037
Finder adressen for en matrix til en karakter og undersøger, hvorvidt den er brugerdefineret eller ej
FØR KALD:
A skal indeholde karakteren, hvis matrix skal findes
VED RETURNERING:
C-flaget sat, hvis karakteren er brugerdefinerbar
C-flaget resat, hvis matricen er placeret i ROM (derved er karakteren ikke brugerdefinerbar)
HL indeholder adressen på matricen
A og øvrige flag ødelægges
Bemærk: Matricen for en karakter består af 8 bytes, hvor den første byte er den øverste linje af karakteren, mens den sidste er den nederste. Bit 7 af en byte er punktet længst til venstre på linjen,

mens bit 0 er punktet længst til højre. Hvis en bit er sat, bliver punktet sat med den ønskede pen, mens en resat bit medfører enten, at punktet sættes med den ønskede baggrund eller at det ikke berøres. Det afhænger af udskrivningstilstanden.

TXT SET MATRIX (31) – BBA8h – 48040
Sætter matricen for en karakter
FØR KALD:
A skal indeholde karakteren, hvis matrix skal sættes
HL skal indeholde adressen på matricen
VED RETURNERING:
C-flaget sat, hvis karakteren er brugerdefinerbar
C-flaget resat, hvis karakteren ikke er brugerdefinerbar
A, BC, DE, HL og andre flag ødelægges
Bemærk: Hvis karakteren ikke er brugerdefinerbar, vil der intet ske. Matricen ændres for alle streams. Ingen karakterer på skærmen bliver dog berørt. Ændringen sker først, når karakteren udskrives næste gang. Se i øvrigt TXT GET MATRIX for forklaring på en matrix.

TXT SET M TABLE (32) – BBABh – 48043
Laver en tabel over brugerdefinerbare karakterer
FØR KALD:
DE skal indeholde nummeret på den første karakter i tabellen
HL skal indeholde startadressen på tabellen
VED RETURNERING:
C-flaget resat, hvis der ingen tabel over brugerdefinerbare karakterer var før
A og HL ødelægges
C-flaget sat, hvis der også før var en tabel over brugerdefinerbare karakterer
A indeholder nummeret på den første karakter i den gamle tabel
HL indeholder startadressen på den gamle tabel
BC, DE og øvrige flag ødelægges
Bemærk: Alle karakterer mellem DE og 255 incl. er hermed at betragte som brugerdefinerbare, og deres matrix overføres til den nye tabel. Hvis DE indeholder mere end 255, vil der ingen brugerdefinerbare karakterer være. Tabellen skal være (256 –

DE) * 8 bytes lang, og karaktererne lægges i denne i rækkefølge. Tabellen skal ligge i RAM, men ikke under ROM. Alle streams benytter den samme tabel over brugerdefinerbare karakterer.

TXT GET M TABLE (33) – BBAEh – 48046
Undersøger adressen på tabellen over brugerdefinerbare karakterer samt angiver den første karakter i denne
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget resat, hvis der ingen tabel over brugerdefinerbare karakterer er
A og HL ødelægges
C-flaget sat, hvis der er en tabel
A indeholder nummeret på den første karakter i tabellen
HL indeholder startadressen på tabellen
Øvrige flag ødelægges
Bemærk: I tabellen er lagt alle karakterer mellem A og 255 incl. En karaktermatrix fylder 8 bytes. Se TXT SET M TABLE for yderligere forklaringer.

TXT GET CONTROLS (34) – BBB1h – 48049
Finder adressen på tabellen over kontrolkoder
FØR KALD:
Ingen krav
VED RETURNERING:
HL indeholder adressen på tabellen over kontrolkoder
Bemærk: I kontrolkodetabellen fylder hver kontrolkode 3 bytes. Først i tabellen står de 3 bytes for kontrolkode 0, så for kode 1 o.s.v. op til kode 31. Den første byte for en kontrolkode angiver antallet af tilhørende parametre. De to sidste bytes angiver adressen på den rutine, som skal udføres (efter alle parametre er modtaget). Denne rutine skal ligge i de midterste 32 K af RAM, og følgende skal være opfyldt:

FØR KONTROLKODEN BRUGES:
A skal indeholde den sidste karakter, der er tilført kontrolkode bufferen

B skal indeholde længden af bufferen incl. kontrolkoden
C skal være lig A
HL skal indeholde adressen på kontrolkode bufferen, idet den peger på kontrolkoden

VED RETURNERING:
AF, BC, DE og HL ødelægges

Kontrolkode bufferen har kun plads til op til 9 parametre. Kontrolkode tabellen initialiseres ved TXT RESET, og den påvirker alle streams.

TXT STR SELECT (35) – BBB4h – 48052
Vælger stream
FØR KALD:
A skal indeholde den ønskede stream
VED RETURNERING:
A indeholder den forrige stream
HL og flagene ødelægges
Bemærk: Stream-nummeret gøres legalt med en AND 7-instruktion. Til en stream hører flg. muligheder: pen, baggrund, markørposition, vinduestørrelse, til- og frakobling af cursor fra såvel bruger som ROM, til- og frakobling af tekstskærm og udskrivningstilstand.

TXT SWAP STREAMS (36) – BBB7h – 48055
Ombytter karakteristika for to streams
FØR KALD:
B skal indeholde det ene streamnummer
C skal indeholde det andet streamnummer
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Stream-numrene legaliseres via AND 7-instruktioner. Ombytningen af de to streams karakteristika omfatter alle de under TXT STR SELECT nævnte.

GRUPPE 3: GRAFIKSKÆRMEN

GRA INITIALISE (1) – BBBAh – 48058
Total initialisering af grafikskærmen som ved start af computeren
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Initialiseringen omfatter bl.a., at den grafiske baggrund sættes til ink 0, den grafiske pen til ink 1, det brugerdefinerbare origo sættes til nederste venstre hjørne, grafikpositionen flyttes hertil, og det grafiske vindue sættes til at dække hele skærmen. Vinduet slettes ikke.

GRA RESET (2) – BBBDh – 48061
Resætter dele af grafikskærmen
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Kun dele initialiseres, og de under GRA INITIALISE omtalte muligheder berøres ikke af denne rutine.

GRA MOVE ABSOLUTE (3) – BBC0h – 48064
Flytter grafikpositionen absolut
FØR KALD:
DE skal indeholde det ønskede x-koordinat
HL skal indeholde det ønskede y-koordinat
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Rutinen flytter grafikpositionen i forhold til det brugerdefinerede origo, d.v.s i absolutte koordinater. Grafikpositionen kan være uden for grafikvinduet.

GRA MOVE RELATIVE (4) – BBC3h – 48067
Flytter grafikpositionen relativt
FØR KALD:
DE skal indeholde det ønskede x-koordinat

HL skal indeholde det ønskede y-koordinat
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Rutinen flytter grafikpositionen i forhold til den nuværende grafikposition, d.v.s. i relative koordinater. Grafikpositionen, hvortil der flyttes, kan være uden for grafikvinduet.

GRA ASK CURSOR (5) – BBC6h – 48070
Undersøger, hvor den nuværende grafikposition er
FØR KALD:
Ingen krav
VED RETURNERING:
DE indeholder x-koordinatet
HL indeholder y-koordinatet
AF ødelægges
Bemærk: Koordinaterne angives i forhold til det selvdefinerede origo.

GRA SET ORIGIN (6) – BBC9h – 48073
Sætter origo i koordinatsystemet og flytter grafikpositionen derhen
FØR KALD:
DE skal indeholde x-koordinatet
HL skal indeholde y-koordinatet
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Koordinaterne skal gives i standardkoordinater, hvor (0,0) er det nederste venstre hjørne af skærmen. Origo er ved start til (0,0), og ved kald af SCR SET MODE retableres dette og et ved kald af denne rutine selvdefineret origo fjernes.

GRA GET ORIGIN (7) – BBCCh – 48076
Undersøger, hvor origo i koordinatsystemet er
FØR KALD:
Ingen krav
VED RETURNERING:
DE indeholder x-koordinatet
HL indeholder y-koordinatet

Bemærk: Koordinaterne, der opgives, er standardkoordinater, hvor (0,0) er skærmens nederste, venstre hjørne

GRA WIN WIDTH (8) – BBCFh – 48079
Sætter venstre og højre kant på grafikvinduet
FØR KALD:
DE skal indeholde koordinatet for den ene kant (stand.)
HL skal indeholde koordinatet for den anden kant (stand.)
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Koordinaterne skal gives i standardkoordinater, hvor (0,0) er skærmens nederste, venstre hjørne. Kanterne flyttes om nødvendigt, så vinduet fylder et helt antal bytes. Ved start fylder vinduet hele skærmen, og det retableres ved kald af SCR SET MODE.

GRA WIN HEIGHT (9) – BBD2h – 48082
Sætter øverste og nederste kant på grafikvinduet
FØR KALD:
DE skal indeholde koordinatet for den ene kant (stand.)
HL skal indeholde koordinatet for den anden kant (stand.)
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Koordinaterne skal angives i standard koordinater, hvor (0,0) er skærmens nederste, venstre hjørne. Vinduet ændres om nødvendigt, så der i vinduet kun indgår hele skærmlinjer. Herved får bunden et lige koordinat, og toppen et ulige. Ved start eller kald af SCR SET MODE sættes vinduet til at fylde hele skærmen.

GRA GET W WIDTH (10) – BBD5h – 48085
Undersøger størrelsen af grafikvinduet horisontalt
FØR KALD:
Ingen krav
VED RETURNERING:
DE indeholder standardkoordinatet for vinduets venstre kant
HL indeholder standardkoordinatet for vinduets højre kant
AF ødelægges

Bemærk: Koordinaterne gives i standardkoordinater, hvor (0,0) er skærmens nederste, venstre hjørne. Koordinaterne er nødvendigvis ikke de samme som ved oprettelsen via GRA WIN WIDTH, da grænserne kan være flyttet, se evt. GRA WIN WIDTH.

GRA GET W HEIGHT (11) – BBD8h – 48088
Undersøger størrelsen af grafikvinduet vertikalt
FØR KALD:
Ingen krav
VED RETURNERING:
DE indeholder standardkoordinatet for vinduets øverste kant
HL indeholder standardkoordinatet for vinduets nederste kant
AF ødelægges
Bemærk: Koordinaterne angives i standard koordinater, hvor (0,0) er skærmens nederste, venstre hjørne. Koordinaterne er nødvendigvis ikke de samme som ved oprettelsen med GRA WIN HEIGHT, da kanterne kan være justeret. Se evt. GRA WIN HEIGHT.

GRA CLEAR WINDOW (12) – BBDBh – 48091
Clear'er grafikvinduet
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Vinduet slettes med den nuværende grafiske baggrund. Grafikpositionen flyttes til det brugerdefinerede origo.

GRA SET PEN (13) – BBDEh – 48094
Sætter den grafiske pen
FØR KALD:
A indeholder den ønskede pen
VED RETURNERING:
AF ødelægges
Bemærk: Den grafiske pen benyttes til plotning af punkter, tegning af linjer og udskrivning af karakterer grafisk. Den ønskede pen legaliseres til den benyttede MODE.

GRA GET PEN (14) – BBE1h – 48097
Undersøger, hvilken grafisk pen der benyttes
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder den benyttede pen
Flagene ødelægges
Bemærk: Den grafiske pen benyttes til at plotte punkter, tegne linjer og udskrive karakterer grafisk.

GRA SET PAPER (15) – BBE4h – 48100
Sætter den grafiske baggrund
FØR KALD:
A indeholder den ønskede baggrund
VED RETURNERING:
AF ødelægges
Bemærk: Baggrunden benyttes ved clearing af grafikvinduet og ved udskrivning af karakterer grafisk. Den ønskede baggrund legaliseres automatisk.

GRA GET PAPER (16) – BBE7h – 48103
Undersøger, hvilken grafisk baggrund der benyttes
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder den benyttede baggrund
Flagene ødelægges
Bemærk: Baggrunden benyttes ved udskrivning af karakterer grafisk og ved clearing af det grafiske vindue.

GRA PLOT ABSOLUTE (17) – BBEAh – 48106
Plotter et punkt absolut
FØR KALD:
DE skal indeholde x-koordinatet
HL skal indeholde y-koordinatet
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Koordinaterne skal angives i absolutte koordinater, d.v.s. i forhold til det selvdefinerede origo. Hvis punktet ligger in-

den for det grafiske vindue, plottes punktet med den grafiske pen i den nuværende grafiske udskrivningstilstand. Hvis punktet ligger udenfor det grafiske vindue, sker der intet.

GRA PLOT RELATIVE (18) – BBEDh – 48109
Plotter et punkt relativt
FØR KALD:
DE skal indeholde x-koordinatet
HL skal indeholde y-koordinatet
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Koordinaterne skal angives i relative koordinater. Hermed menes i forhold til den nuværende grafikposition. Hvis punktet ligger inden for det grafiske vindue, plottes punktet med den grafiske pen i den nuværende grafiske udskrivningstilstand. Hvis punktet ligger uden for det grafiske vindue, sker der intet.

GRA TEST ABSOLUTE (19) – BBF0h – 48112
Tester et punkts ink
FØR KALD:
DE skal indeholde x-koordinatet
HL skal indeholde y-koordinatet
VED RETURNERING:
A indeholder punktets ink (eller baggrundsfarven)
BC, DE, HL og flagene ødelægges
Bemærk: Koordinatet skal angives i absolutte koordinater, altså i forhold til det brugerdefinerede origo. Hvis punktet ligger inden for det grafiske vindue, dekodes dets farve til et ink-nummer, der lægges i A. Hvis punktet ligger uden for det grafiske vindue, returneres med den grafiske baggrund.

GRA TEST RELATIVE (20) – BBF3h – 48115
Tester et punkts ink
FØR KALD:
DE skal indeholde x-koordinatet
HL skal indeholde y-koordinatet
VED RETURNERING:
A indeholder punktets ink (eller baggrundsfarven)
BC, DE, HL og flagene ødelægges

Bemærk: Koordinaterne skal angives i relative koordinater, d.v.s. i forhold til grafikpositionen. Hvis punktet ligger inden for det grafiske vindue, dekodes dets farve til et ink-nummer, der placeres i A-registret. Hvis punktet ligger uden for det grafiske vindue, returneres med den grafiske baggrund.

GRA LINE ABSOLUTE (21) – BBF6h – 48118
Tegner en streg absolut
FØR KALD:
DE skal indeholde endepunktets x-koordinat
HL skal indeholde endepunktets y-koordinat
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Endepunktets koordinater skal angives i absolutte koordinater, altså i forhold til det selvdefinerede origo. Alle punkter mellem grafikpositionen og endepunktet inden for det grafiske vindue plottes med den grafiske pen i den grafiske udskrivningstilstand. Alle punkter uden for det grafiske vindue plottes ikke. Grafikpositionen rykkes til endepunktet, uanset dets position.

GRA LINE RELATIVE (22) – BBF9h – 48121
Tegner en streg relativt
FØR KALD:
DE skal indeholde endepunktets x-koordinat
HL skal indeholde endepunktets y-koordinat
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Endepunktets koordinater skal angives i relative koordinater, d.v.s. i forhold til den nuværende grafikposition. Alle punkter mellem grafikpositionen og endepunktet inden for det grafiske vindue plottes med den grafiske pen i den grafiske udskrivningstilstand. Alle punkter uden for det grafiske vindue plottes ikke. Grafikpositionen rykkes til endepunktet, uanset dets position.

GRA WR CHAR (23) – BBFCh – 48124
Udskriver en karakter på den nuværende grafikposition
FØR KALD:
A skal indeholde karakteren

VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Karakteren bliver udskrevet med sit øverste, venstre hjørne på den nuværende grafikposition. Efter udskrivningen flyttes grafikpositionen til højre: 32 punkter i mode 0, 16 i mode 1 og 8 punkter i mode 2. Alle karakterer, også kontrolkoder, udskrives. Udskriften foretages med grafisk pen og altid på den grafiske baggrund, udskrivningstilstanden for tekstskærmen underordnet.

GRUPPE 4: SKÆRMEN

SCR INITIALISE (1) – BBFFh – 48127
Initialiserer skærmen som ved start af computeren
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Initialiseringen omfatter bl.a. retablering af de originale inks, perioder til blink sættes til deres startværdier, skærmen sættes i mode 1, skærmen placeres i området C000h-FFFFh, skærmen slettes til ink 0 og den grafiske udskrivningstilstand bliver FORCE MODE.

SCR RESET (2) – BC02h – 48130
Resætter dele af skærmen
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Rutinen resætter kun dele af skærmen. Heriblandt kan nævnes retablering af de oprindelige inks og deres blinkperioder samt den grafiske udskrivningstilstand, der sættes til FORCE MODE.

SCR SET OFFSET (3) – BC05h – 48133
Sætter off-set for den første karakter på skærmen
FØR KALD:
HL skal indeholde det ønskede off-set
VED RETURNERING:
AF og HL ødelægges
Bemærk: Værdien i HL legaliseres til højst at være 07FEh. Off-set bruges ved scroll og ved udregning af skærmpositioner. Det bliver sat til nul ved clearing eller skift af MODE.

SCR SET BASE (4) – BC08h – 48136
Sætter startadressen for skærmbilledet
FØR KALD:
A skal indeholde MSB af startadressen
VED RETURNERING:
AF og HL ødelægges
Bemærk: Da skærmen fylder 16 Kbytes, vil værdien i A blive legaliseret via AND 192. Startadressen benyttes til udregning af skærmpositioner.

SCR GET LOCATION (5) – BC0Bh – 48139
Undersøger, hvor skærmbilledets startadresse er samt dets offset.
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder MSB af skærmbilledets startadresse
HL indeholder det nuværende off-set
Flagene ødelægges
Bemærk: Værdierne behøver ikke være lig de, som indtastedes ved SCR SET OFFSET eller SCR SET BASE, da disse er legaliseret. Desuden ændres off-set ved scroll.

SCR SET MODE (6) – BC0Eh – 48142
Sætter skærmen i MODE
FØR KALD:
A indeholder den ønskede tilstand (MODE)
VED RETURNERING:
AF, BC, DE og HL ødelægges

Bemærk: Den ønskede MODE legaliseres via AND 3. Hvis resultatet bliver 3, ændres intet. De 3 tilstande har følgende karakteristika:

MODE 0: 160x200 punkter – 20x25 karakterer
MODE 1: 320x200 punkter – 40x25 karakterer
MODE 2: 640x200 punkter – 80x25 karakterer

Ved kald af denne rutine slettes skærmen, og alle tekst og grafik vinduer er sat til hele skærmen. Origo sættes til nederste, venstre hjørne og penne og baggrunde gøres lovlige (ved skift til MODE med færre farvemuligheder ændres de krævede farver).

SCR GET MODE (7) – BC11h – 48145
Undersøger, hvilken MODE skærmen er i
FØR KALD:
Ingen krav
VED RETURNERING:
MODE 0: C-flaget sat
Z-flaget resat
MODE 1: C-flaget resat
Z-flaget sat
MODE 2: C-flaget resat
Z-flaget resat
A indeholder MODE-nummeret
Flagene ødelægges
Bemærk: De forskellige tilstandes karakteristika kan iagttages under SCR SET MODE.

SCR CLEAR (8) – BC14h – 48148
Clear'er hele skærmen til ink 0
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Skærmen slettes til ink 0, hvad enten der benyttes denne ink som baggrundsfarve eller ej. Off-set sættes til nul.

SCR CHAR LIMITS (9) – BC17h – 48151
Undersøger skærmens størrelse m.h.t. karakterer

FØR KALD:
Ingen krav
VED RETURNERING:
B indeholder den sidste kolonne på skærmen
C indeholder den sidste række på skærmen
AF ødelægges
Bemærk: Kolonne- og rækkenumrene gives i forhold til øverste venstre hjørne, der har koordinaterne (0,0). Den sidste kolonne vil således være 19 i mode 0, 39 i mode 1 og 79 i mode 2, mens den sidste række altid vil være 24.

SCR CHAR POSITION (10) – BC1Ah – 48154
Konverterer en karakterposition til en skærmadresse
FØR KALD:
H skal indeholde den ønskede kolonne
L skal indeholde den ønskede række
VED RETURNERING:
HL indeholder skærmadressen på det øverste, venstre hjørne af karakteren
B indeholder bredden af én karakter i bytes
AF ødelægges
Bemærk: Koordinaterne skal angives i forhold til skærmens øverste venstre hjørne, der har koordinaterne (0,0). Karakterpositionen undersøges ikke, hvorved en ulovlig vil medføre en meningsløs skærmadresse. Denne udregnes ved hjælp af skærmbilledets startadresse, off-set samt karakterens bredde.

SCR DOT POSITION (11) – BC1Dh – 48157
Konverterer en grafikposition til en skærmadresse
FØR KALD:
DE skal indeholde x-koordinatet
HL skal indeholde y-koordinatet
VED RETURNERING:
HL indeholder skærmadressen for det angivne punkt
C indeholder en angivelse af, hvor i byten punktet er placeret
B indeholder én mindre end antallet af punkter i en byte
AF og DE ødelægges
Bemærk: Koordinaterne skal angives i forhold til nederste venstre hjørne, der har koordinaterne (0,0). Hvert koordinat svarer

til ét punkt. Grafikpositionen undersøges ikke, hvorfor en ulovlig position vil medføre en meningsløs skærmadresse. Skærmadressen udregnes via skærmbilledets startadresse, off-set samt hvor mange punkter én byte indeholder.

SCR NEXT BYTE (12) – BC20h – 48160
Udregner skærmadressen på byten til højre
FØR KALD:
HL skal indeholde en skærmadresse
VED RETURNERING:
HL indeholder skærmadressen på byten til højre
AF ødelægges
Bemærk: Hvis den næste byte er uden for linjen, vil der findes adressen på den næste byte i blokken (bestående af 2048 bytes). Dette vil normalt være 8 linjer længere nede. Hvis byten er efter den sidste i en blok, fås adressen på den første af de 48 ikke viste bytes. Rutinen bruges bl.a. ved tegning af linjer.

SCR PREV BYTE (13) – BC23h – 48163
Udregner skærmadressen på byten til venstre
FØR KALD:
HL skal indeholde en skærmadresse
VED RETURNERING:
HL indeholder skærmadressen på byten til venstre
AF ødelægges
Bemærk: Hvis den foregående byte er uden for linjen, vil den findes på adressen på den foregående byte i blokken. Dette vil normalt være den sidste byte i linjen 8 linjer højere oppe. Hvis byten ligger før den første i en blok, fås adressen på den sidste af de 48 ikke viste bytes. Rutinen minder om SCR NEXT BYTE, se derfor evt. denne.

SCR NEXT LINE (14) – BC26h – 48166
Udregner skærmadressen på byten nedenunder
FØR KALD:
HL skal indeholde en skærmadresse
VED RETURNERING:
HL indeholder skærmadressen på byten nedenunder
AF ødelægges

Bemærk: Hvis denne byte er uden for skærmen, vil adressen, der returneres, være meningsløs. Se evt. SCR NEXT BYTE eller SCR PREV BYTE.

SCR PREV LINE (15) – BC29h – 48169
Udregner skærmadressen på byten ovenover
FØR KALD:
HL skal indeholde en skærmadresse
VED RETURNERING:
HL indeholder skærmadressen på byten ovenover
AF ødelægges
Bemærk: Hvis denne byte er oven over skærmen, vil den adresse, der returneres, være meningsløs. Se evt. SCR NEXT BYTE eller SCR PREV BYTE.

SCR INK ENCODE (16) – BC2Ch – 48172
Afkoder en ink til farvekoden
FØR KALD:
A skal indeholde ink-nummeret
VED RETURNERING:
A indeholder farvekoden
Flagene ødelægges
Bemærk: I mode 2 er der 8 punkter i en byte, hvorfor hvert punkt kun har 2 farvemuligheder. I mode 1 er der 4 punkter, og hvert punkt har 4 farvemuligheder. Endelig er der mode 0, hvor der er 2 punkter i en byte – hvert punkt har 16 farvemuligheder.
Mode 2:

Punktet længst til venstre:	bit 7 i byten
Punktet næstlængst t.v.:	bit 6 i byten
	bit 5 i byten
	bit 4 i byten
	bit 3 i byten
	bit 2 i byten
Punktet næstlængst t.h.:	bit 1 i byten
Punktet længst til højre:	bit 0 i byten

Mode 1:

Punktet længst til venstre:	bit 3 og bit 7 i byten
Punktet næstlængst t.v.:	bit 2 og bit 6 i byten
Punktet næstlængst t.h.:	bit 1 og bit 5 i byten

Punktet længst til højre: bit 0 og bit 4 i byten
Mode 0:
Punktet til venstre: bittene 1, 5, 3 og 7
Punktet til højre: bittene 0, 4, 2 og 6
Punkternes farve er altså sammensat på ovenstående måde. Dette bruges bl.a. til at finde farven af ét enkelt punkt.

SCR INK DECODE (17) – BC2Fh – 48175
Dekoder en farvekode til et ink-nummer
FØR KALD:
A skal indeholde farvekoden, der skal dekodes
VED RETURNERING:
A indeholder ink-nummeret
Flagene ødelægges
Bemærk: Rutinen dekoder en farvekode til en ink. Dette har betydning ved de enkelte punkters farve. Der henvises til SCR INK ENCODE.

SCR SET INK (18) – BC32h – 48178
Sætter hvilke farver en ink er
FØR KALD:
A skal indeholde ink-nummeret
B skal indeholde den første farve
C skal indeholde den anden farve
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Hvis de to farver er forskellige, vil ink'en skifte mellem disse. Tiden imellem disse skift kan sættes med SCR SET FLASHING. Hvis de to farver er ens, vil ink'en antage den samme farve hele tiden. Ink-nummeret legaliseres via AND 15, mens farvenumrene lovliggøres med AND 31. Farverne fra 27 til 31 er kopier af andre farver.

SCR GET INK (19) – BC35h – 48181
Undersøger, hvilke to farver en ink skifter imellem
FØR KALD:
A skal indeholde ink-nummeret
VED RETURNERING:
B indeholder den første farve

C indeholder den anden farve
AF, DE og HL ødelægges
Bemærk: Ink-nummeret legaliseres med AND 15.

SCR SET BORDER (20) – BC38h – 48184
Sætter skærmkantens farver
FØR KALD:
B skal indeholde den første farve
C skal indeholde den anden farve
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Farverne legaliseres med AND 31. Farverne fra 27 til 31 er ikke brugbare – de er blot kopier af andre farver. Den tid, der går mellem skiftene, kan sættes med SCR SET FLASHING. Hvis de to farver er ens, vil skærmkanten kun antage én farve.

SCR GET BORDER (21) – BC3Bh – 48187
Undersøger skærmkantens farver
FØR KALD:
Ingen krav
VED RETURNERING:
B indeholder den første farve
C indeholder den anden farve
AF, DE og HL ødelægges.

SCR SET FLASHING (22) – BC3Eh – 48190
Sætter blink-perioder
FØR KALD:
H skal indeholde tiden for den første farve
L skal indeholde tiden for den anden farve
VED RETURNERING:
AF og HL ødelægges
Bemærk: Blink-perioderne skal angives i antallet af frame flybacks. Af disse forekommer 50 i sekundet. Med en angivelse af 0 menes 256. Ved start er perioderne sat til 10 (1/5 sek.). Blinkene påvirker alle inks samt skærmkanten.

SCR GET FLASHING (23) – BC41h – 48193
Undersøger blink-perioderne

FØR KALD:
Ingen krav
VED RETURNERING:
H indeholder tiden for den første farve
L indeholder tiden for den anden farve
AF ødelægges
Bemærk: Blink-perioderne vedrører alle inks samt skærmkanten. Angivelserne er i antal frame flybacks, af hvilke der forekommer 50 i sekundet. En angivelse på 0 er ensbetydende med 256.

SCR FILL BOX (24) – BC44h – 48196
Fylder et område af skærmen med en ink
FØR KALD:
A skal indeholde farvekoden, som området skal fyldes med
H skal indeholde koordinatet på områdets venstre kolonne
D skal indeholde koordinatet på områdets højre kolonne
L skal indeholde koordinatet på områdets øverste række
E skal indeholde koordinatet på områdets nederste række
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Koordinaterne skal angives i forhold til skærmens øverste, venstre hjørne, der har koordinaterne (0,0). Koordinaterne undersøges ikke, hvorfor illegale værdier kan medføre uforudsigelige hændelser. Området fyldes underordnet udskrivningstilstanden.

SCR FLOOD BOX (25) – BC47h – 48199
Fylder et område af skærmen med en ink
FØR KALD:
C skal indeholde farvekoden, som området skal fyldes med
HL skal indeholde skærmadressen på områdets øverste, venstre hjørne
D skal indeholde bredden af området angivet i bytes
E skal indeholde højden af området angivet i skærmlinjer
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Området skal være rektangulært, og kanterne af det skal ligge på en givet byte for at kunne angives. Hvis dele af rektanglet ligger uden for skærmen, kan der ske uforudsigelige

hændelser. Området fyldes uden hensyn til udskrivningstilstanden.

SCR CHAR INVERT (26) – BC4Ah – 48202
Inverterer karakteren på en position
FØR KALD:
B skal indeholde den ene farvekode
C skal indeholde den anden farvekode
H skal indeholde kolonnenummeret
L skal indeholde rækkenummeret
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Alle punkter på positionen, som er skrevet med den ene farvekode, skrives med den anden, og vice versa. Derved fås en form for invertering. Numrene på positionen skal angives i forhold til skærmens øverste, venstre hjørne, der har (0,0). En ulovlig position vil medføre uforudsigelige ting, da den ikke undersøges. Punkter i karakteren på den givne position, der ikke er sat med en af de to angivne farvekoder, vil også ændre farve efter en XOR-instruktion. Kald af rutinen to gange vil retablere de oprindelige farver.

SCR HW ROLL (27) – BC4Dh – 48205
Scroller hele skærmen en karakter op eller ned
FØR KALD:
B skal indeholde 0, hvis skærmen skal rulle ned
B skal være forskellig fra 0, hvis skærmen skal rulle opad
A skal indeholde den farvekode, som den nye linje skal clear'es til
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Skærmen flyttes 8 linjer op eller ned (en række). Dette er ensbetydende med en hel karakter. Skærmen scrolles ved at ændre off-set. Ved scroll opad forøges dette med 80, og den nederste linje slettes med den angivne farvekode. Ved scroll nedad, decrementeres off-set med 80, og den øvrige linje slettes med den angivne farvekode. Den såkaldte »roll count« – se TXT GET CURSOR – berøres ikke.

SCR SW ROLL (28) – BC50h – 48208
Scroller en del af skærmen op eller ned
FØR KALD:
B skal indeholde 0, hvis området skal rulles ned
B skal være forskellig fra 0, hvis området skal rulles op
A skal indeholde den farvekode, som den nye linje skal antage
H skal indeholde nummeret på den venstre kolonne
D skal indeholde nummeret på den højre kolonne
L skal indeholde nummeret på den øverste række
E skal indeholde nummeret på den nederste række
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Kolonne- og rækkenumrene skal angives i forhold til skærmens øverste, venstre hjørne, der har (0,0). Numrene undersøges ikke, hvorfor ulovlige angivelser kan medføre uforudsigelige hændelser. Ved scroll opad bliver den nederste linje slettet med den angivne farvekode, mens ved scroll nedad bliver den øverste linje slettet med denne. Den såkaldte »roll count« – se TXT GET CURSOR – berøres ikke.

SCR UNPACK (29) – BC53h – 48211
Udvider karaktermatrix i den nuværende MODE
FØR KALD:
HL skal indeholde adressen på en matrix
DE skal indeholde adressen på et område, hvori udvidelsen lægges
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Hvert byte i matricen konverteres til 1 byte i mode 2, 2 bytes i mode 1 og 4 bytes i mode 0. Området, hvori udvidelsen skal lægges, skal således være 8, 16 eller 32 bytes lang, afhængig af den nuværende MODE.

SCR REPACK (30) – BC56h – 48214
Konverterer en karakter til en matrix
FØR KALD:
A skal indeholde den farvekode, som der skal sammenlignes med
H skal indeholde kolonnenummeret

L skal indeholde rækkenummeret
DE skal indeholde startadressen på det område, hvori matricens udseende skal placeres
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Hvert punkt i karakteren sammenlignes med den i A beliggende farvekode. Hvis punktet har denne ink, sættes bitten i matricen, ellers resættes den. Karakteren er den, der står på den angivne position. Denne er givet i kolonne- og rækkenummer i forhold til skærmens øverste, venstre hjørne, der har (0,0). Positionen kontrolleres ikke, hvorfor ulovlige positioner kan medføre uforudsigelige hændelser. Den fundne matrix fylder som sædvanlig 8 bytes og er helt normal (for dens udseende, se evt. TXT SET MATRIX).

SCR ACCESS (31) – BC59h – 48217
Sætter den grafiske udskrivningstilstand
FØR KALD:
A skal indeholde den ønskede tilstand
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Der er 4 muligheder for den grafiske udskrivningstilstand, der har at gøre med bl.a. plotning og tegning af linjer. De 4 muligheder er:

0:	FORCE MODE	Punktet får den nye ink
1:	XOR MODE	Punktet får ink (ny XOR gammel)
2:	AND MODE	Punktet får ink (ny AND gammel)
3:	OR MODE	Punktet får ink (ny OR gammel)

Ved start af computeren er udskrivningstilstanden FORCE MODE, hvorved alle punkter overtegnes, når nye punkter skal tegnes. De tre øvrige tilstande opererer alle med de enkelte bit i de angivne inks. Ved AND og OR sættes og resættes de enkelte bits i punktet, hvor der skal tegnes. Ved XOR MODE vil tegning af det samme punkt to gange medføre, at det gamle punkt træder frem igen.
Eksempel:
Der plottes oven på ink 1 med ink 2

0: FORCE MODE: Det nye punkt får ink 2
1: XOR MODE: Det nye punkt får ink 3 (1 XOR 2)
2: AND MODE: Det nye punkt får ink 0 (1 AND 2)
3: OR MODE: Det nye punkt får ink 3 (1 OR 2)

Nummeret på tilstanden legaliseres ved AND 3.

SCR PIXELS (32) – BC5Ch – 48220
Skriver et punkt på skærmen underordnet den grafiske udskrivningstilstand
FØR KALD:
B skal indeholde farvekoden, hvormed der skrives
C skal indeholde en angivelse af, hvor i byten koden skal placeres
HL skal indeholde skærmadressen for punktet
VED RETURNERING:
AF ødelægges
Bemærk: Skærmadressen kontrolleres ikke, så en ulovlig kan have uforudsigelige virkninger. Punktet farves med den angivne farvekode den grafiske udskrivningstilstand underordnet, hvilket svarer til FORCE MODE.

SCR HORIZONTAL (33) – BC5Fh – 48223
Tegner en vandret linje
FØR KALD:
A skal indeholde farvekoden, som linjen skal tegnes med
DE skal indeholde x-koordinatet på starten af linjen
BC skal indeholde x-koordinatet på slutningen af linjen
HL skal indeholde y-koordinatet
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Linjen tegnes helt vandret, og den tegnes i den grafiske udskrivningstilstand. Koordinaterne skal angives i forhold til skærmens nederste, venstre hjørne, der har koordinaterne (0,0). X-koordinatet på starten af linjen skal være mindre end eller lig med x-koordinatet på slutningen. Hvis der angives koordinater, der er ulovlige, kan der ske uforudsigelige hændelser, da de ikke undersøges.

SCR VERTICAL (34) – BC62h – 48226
Tegner en lodret linje
FØR KALD:
A skal indeholde farvekoden, som linjen skal tegnes med
DE skal indeholde x-koordinatet
HL skal indeholde y-koordinatet på starten af linjen
BC skal indeholde y-koordinatet på slutningen af linjen
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Linjen tegnes helt lodret, og den tegnes i den grafiske udskrivningstilstand. Koordinaterne skal angives i forhold til skærmens nederste, venstre hjørne, der har koordinaterne (0,0). Y-koordinatet på starten af linjen skal være mindre end eller lig med y-koordinatet på slutningen. Hvis der angives koordinater, der er ulovlige, kan der ske uforudsigelige hændelser, da de ikke kontrolleres.

GRUPPE 5: KASSETTE-BÅNDOPTAGEREN

CAS INITIALISE (1) – BC65h – 48229
Initialisering af kassettebåndoptageren som ved start af computeren
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Initialiseringen omfatter lukning af alle streams, retablering af den oprindelige overførelseshastighed og tilladelse til udskrivning af startmeddelelser.

CAS SET SPEED (2) – BC68h – 48232
Sætter overførelseshastighed
FØR KALD:

HL skal indeholde en værdi til bestemmelse af overførelseshastigheden
A skal indeholde en værdi til at skelne mellem de overførte bit
VED RETURNERING:
AF og HL ødelægges
Bemærk: Værdien, der placeres i HL, skal ligge inden for intervallet 130-480 mikrosekunder. Den gennemsnitlige overførelseshastighed kan udregnes som 1.000.000/(3 * HL), hvilket igen betyder, at den er 333.333/HL. Dette giver gennemsnitsværdier mellem 694 og 2564 baud. Der kan kun være tale om gennemsnitsværdier, da en sat bit fylder dobbelt så meget som en resat. Angivne værdier uden for intervallet 130-480 vil medføre en fejl. Værdien i A angiver, hvor mange mikrosekunder der skal hhv. tillægges en sat bit og fratrækkes en resat i specielle tilfælde. Denne værdi skal ligge mellem 0 og 255. Ved start af computeren er værdien til bestemmelse af overførelseshastighed 333 (1000 baud) og værdien til skelnen 25. Ellers benyttes 167 (2000 baud) og 50 som skelnen. Disse to sæt værdier har været afprøvet og er fundet de mest velegnede af Amstrad-producenten.

CAS NOISY (3) – BC6Bh – 48235
Tillader eller forhindrer udskrivning af start- og informationsmeddelelser.
FØR KALD:
A skal indeholde 0, hvis meddelelser skal tillades
A skal være forskellig fra 0, hvis meddelelser skal forhindres
VED RETURNERING:
AF ødelægges
Bemærk: Ved frakobling forhindres flg. meddelelser:
Press PLAY then any key:
Press REC and PLAY then any key:
Found FILENAME block N
Loading FILENAME block N
Saving FILENAME block N
Fejlmeddelelserne omfattende følgende kan ikke forhindres:
Read error x
Write error a
Rewind tape

CAS START MOTOR (4) – BC6Eh – 48238
Starter kassettebåndoptagerens motor
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis operationen er lykkedes
C-flaget resat, hvis brugeren har trykket ESC
A indeholder den foregående motorstatus
Øvrige flag ødelægges
Bemærk: Status fra før denne rutines kald kan udnyttes af CAS RESTORE MOTOR.

CAS STOP MOTOR (5) – BC71h – 48241
Stopper kassettebåndoptagerens motor
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis operationen lykkes
C-flaget resat, hvis brugeren har trykket ESC
A indeholder den foregående motorstatus
Øvrige flag ødelægges
Bemærk: Status fra før denne rutines kald kan udnyttes af CAS RESTORE MOTOR.

CAS RESTORE MOTOR (6) – BC74h – 48244
Genetablerer den forrige motorstatus
FØR KALD:
A skal indeholde den forrige status af motor
VED RETURNERING:
C-flaget sat, hvis operationen er lykkedes
C-flaget resat, hvis brugeren har trykket ESC
A og de øvrige flag ødelægges
Bemærk: Denne rutine genetablerer den status af kassettebånd-optagerens motor, som var før kald af CAS START MOTOR eller CAS STOP MOTOR. Status er returneret af disse rutiner.

CAS IN OPEN (7) – BC77h – 48247
Åbner en fil til indlæsning af blok

FØR KALD:
B skal indeholde længden af filnavnet
HL skal indeholde adressen på filnavnet
DE skal indeholde adressen på en 2K buffer
VED RETURNERING:
Åbningen er lykkedes
C-flaget sat
Z-flaget resat
HL indeholder adressen på en buffer indeholdende 64 bytes om filen
DE indeholder en adresse, der viser, hvor de indlæste data var placeret oprindelig
BC indeholder længden af filen angivet i bytes
A indeholder filtypen
Stream'en er i brug i forvejen
C-flaget resat
Z-flaget resat
A, BC, DE og HL ødelægges
Brugeren har trykket ESC
C-flaget resat
Z-flaget sat
A, BC, DE og HL ødelægges
IX og de øvrige flag ødelægges
Bemærk: 2 K bufferen angivet i DE ved kald skal bruges til at gemme den indlæste blok i. Blokken vil blive i dette område, indtil filen lukkes med CAS IN CLOSE eller CAS IN ABANDON. Bufferen må gerne ligge under ROM i RAM. Filnavnet skal være på 16 karakterer. Er det længere, bliver det beskåret, mens det udfyldes med 00h, hvis det ikke er på 16 tegn. Små bogstaver ændres til store. Hvis fillængden er på 0 eller navnet starter med dette, vil den næste fil på båndet blive indlæst i stedet. Bufferen, der indeholder besked om filen, vil ligge i de midterste, centrale 32K af RAM. Området har dette udseende:

bytes 0-15:	filnavn
byte 16:	bloknummer (den første har nummer 1)
byte 17:	hvis denne værdi er forskellig fra nul, er dette filens sidste blok
byte 18:	filtypen – se herunder

byte 19-20:	antallet af bytes i optagelsen
byte 21-22:	adressen, hvor de indlæste data oprindelig var placeret
byte 23:	hvis denne værdi er forskellig fra nul, er dette filens første blok
byte 24-25:	filens totale længde i bytes
byte 26-27:	udførelsesadresen for maskinkodeprogrammør
byte 28-63:	ubenyttet – kan bruges af programmøren

byte 18 har flg. udseende:

bit 0:	sat, hvis filen er beskyttet
bit 1-3:	0 = BASIC
	1 = binær
	2 = skærmbillede
	3 = ASCII fil
	4-7 = ubenyttet
bit 4-7:	0 = alle file undtaget ASCII fil
	1 = ASCII fil

Brugeren må kun læse fra dette område.

CAS IN CLOSE (8) – BC7Ah – 48250
Lukker en fil til indlæsning
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis lukningen er lykkedes
C-flaget resat, hvis stream'en var lukket i forvejen
A, BC, DE, HL og øvrige flag ødelægges
Bemærk: Denne rutine skal benyttes til at lukke en fil efter at have læst fra denne med CAS IN CHAR eller CAS IN DIRECT.

CAS IN ABANDON (9) – BC7Dh – 48253
Opgiver at læse fra en fil og lukker stream'en
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges

Bemærk: Denne rutine til lukning af en fil skal benyttes, hvis der er opstået en fejl ved indlæsningen.

CAS IN CHAR (10) – BC80h – 48256
Indlæser en karakter fra en fil
FØR KALD:
Ingen krav
VED RETURNERING:
Karakteren er indlæst
  C-flaget sat
  Z-flaget resat
  A indeholder den indlæste karakter
Enden af filen er nået
  C-flaget resat
  Z-flaget resat
  A ødelægges
Brugeren har trykket ESC
  C-flaget resat
  Z-flaget sat
  A ødelægges
IX og øvrige flag ødelægges
Bemærk: Hvis brugeren tidligere har presset ESC eller stream'en ikke er åben, vil det blive angivet som enden på filen. Når denne metode til indlæsning er valgt, kan indlæsning af en hel fil (se CAS IN DIRECT) ikke vælges. Denne rutine skal benyttes til indlæsning af tekstfiler.

CAS IN DIRECT (11) – BC83h – 48259
Indlæser en fil i hukommelsen
FØR KALD:
HL skal indeholde adressen, hvor filen skal lagres
VED RETURNERING:
Filen er indlæst
  C-flaget sat
  Z-flaget resat
  HL indeholder startadressen for maskinkodeprogram
Filen var ikke åbnet
  C-flaget resat
  Z-flaget resat

HL ødelægges
Brugeren har trykket ESC
C-flaget resat
Z-flaget sat
HL ødelægges
A, BC, DE, IX og øvrige flag ødelægges
Bemærk: Stream'en skal være åben, inden denne rutine kaldes. Når denne metode til indlæsning er valgt, kan indlæsning af en fil karakter for karakter (se CAS IN CHAR) ikke anvendes. Filen skal indlæses et hvilket som helst sted i RAM. Hvis filen er indlæst, indeholder HL det sted, hvor programmet skal udføres fra. Det er byte 26-27 i den buffer, som omtales under CAS IN OPEN. Denne metode til indlæsning skal benyttes ved indlæsning af maskinkoderutiner.

CAS RETURN (12) – BC86h – 48262
Lægger den sidst indlæste karakter tilbage i filen
FØR KALD:
Ingen krav
VED RETURNERING:
Alt bevares
Bemærk: Den sidst indlæste karakter vil således være den næste, der bliver indlæst af CAS IN CHAR.

CAS TEST EOF (13) – BC89h – 48265
Undersøger, om enden af filen er nået
FØR KALD:
Ingen krav
VED RETURNERING:
Enden af filen blev ikke nået
C-flaget sat
Z-flaget resat
Enden af filen blev nået
C-flaget resat
Z-flaget resat
Brugeren har trykket ESC
C-flaget resat
Z-flaget sat
A, IX og øvrige flag ødelægges.

Bemærk: Denne rutine sætter stream'en i den tilstand, der indlæser karakterer. Indlæsning af hele file er ikke muligt efter kald af denne rutine. CAS RETURN kan ikke kaldes umiddelbart efter denne, da en karakter først skal indlæses.

CAS OUT OPEN (14) – BC8Ch – 48268
Åbner en fil for udlæsning
FØR KALD:
B skal indeholde filnavnets længde
HL skal indeholde adressen på filnavnet
DE skal indeholde adressen på en 2K buffer
VED RETURNERING:
Åbningen er lykkedes
  C-flaget sat
  Z-flaget resat
  HL indeholder en adresse på en buffer indeholdende 64 bytes om filen
Stream'en er i brug i forvejen
  C-flaget resat
  Z-flaget resat
  HL ødelægges
Brugeren har trykket ESC
  C-flaget resat
  Z-flaget sat
  HL ødelægges
A, BC, DE, IX og øvrige flag ødelægges
Bemærk: 2 K bufferen angivet i DE ved kald skal bruges til at gemme den blok, der skal udlæses, i. Blokken vil blive i dette område, indtil filen lukkes ved CAS OUT CLOSE eller CAS OUT ABANDON. Bufferen skal ligge i RAM – gerne under ROM. Filnavnet skal være på 16 karakterer. Længere navne vil blive beskåret, mens navne på færre tegn vil blive udfyldt med 00h. Små bogstaver ændres til store. Når filen åbnes, indeholder HL ved returnering en adresse,der angiver et område indeholdende 64 bytes om filen. For den fulde forklaring omkring denne buffer, se CAS IN OPEN. Systemet skriver selv i disse bytes, men brugeren må skrive til dele af dem. De tilladte er filtypen (byte 18), filens længde (byte 24-25), udførelsesadresse (byte 26-27) og hele bru-

geområdet (byte 28-63). Disse fyldes med 00h, hvis brugeren ikke benytter dem. Andre end de her nævnte dele må ikke berøres.

CAS OUT CLOSE (15) – BC8F – 48271
Lukker en fil til udlæsning
FØR KALD:
Ingen krav
VED RETURNERING:
Lukningen er lykkedes
  C-flaget sat
  Z-flaget resat
Stream'en var ikke åbnet i forvejen
  C-flaget resat
  Z-flaget resat
Brugeren har trykket ESC
  C-flaget resat
  Z-flaget sat
A, BC, DE, HL, IX og øvrige flag ødelægges
Bemærk: Denne rutine skal kaldes, når den sidste blok af data er blevet udlæst via enten CAS OUT CHAR eller CAS OUT DIRECT. Hvis brugeren trykker ESC under udlæsningen af den sidste blok, forbliver filen åben.

CAS OUT ABANDON (16) – BC92h – 48274
Lukker udlæsningen af en fil øjeblikkelig
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Denne rutine lukker en fil med det samme. Data, der ikke er udlæst, bliver det heller ikke. Rutinen kan benyttes, hvis der opstår fejl under udlæsningen.

CAS OUT CHAR (17) – BC95h – 48277
Udlæser en karakter
FØR KALD:
A skal indeholde den karakter, der skal udlæses
VED RETURNERING:
Karakteren er udlæst

C-flaget sat
Z-flaget resat
Filen er ikke åbnet
C-flaget resat
Z-flaget resat
Brugeren har trykket ESC
C-flaget resat
Z-flaget sat
A, IX og øvrige flag ødelægges.
Bemærk: Hvis brugeren tidligere har tykket ESC eller filen er blevet udlæst via CAS OUT DIRECT, vil det være angivet som ikke åbnet fil. Når alle karakterer er udlæst, skal CAS OUT CLOSE kaldes. Når denne metode til udlæsning er valgt, kan udlæsning af en hel fil (se CAS OUT DIRECT) ikke anvendes.

CAS OUT DIRECT (18) – BC98h – 48280
Udlæser en hel fil fra hukommelsen
FØR KALD:
HL skal indeholde adressen på de data, som skal udlæses
DE skal indeholde antallet af data (bytes)
BC skal indeholde startadressen for eksekvering
A skal indeholde filtypen
VED RETURNERING:
Filen er udlæst
C-flaget sat
Z-flaget resat
Filen var ikke åbnet
C-flaget resat
Z-flaget resat
Brugeren har trykket ESC
C-flaget resat
Z-flaget sat
A, BC, DE, HL, IX og øvrige flag ødelægges
Bemærk: Når denne metode til udlæsning er valgt, kan udlæsning af en fil karakter for karakter (se CAS OUT CHAR) ikke benyttes. Efter udlæsningen skal CAS OUT CLOSE kaldes for at lukke stream'en. Ved kaldet skal BC indeholde startadressen, der er byte 26-27 i den buffer, som omtales under CAS IN OPEN. A skal indeholde filtypen, der er byte 18 i samme buffer.

CAS CATALOG (19) – BC9Bh – 48283
Laver et katalog over de på båndet indlæste blokke
FØR KALD:
DE skal indeholde adressen på en 2K buffer
VED RETURNERING:
Katalogiseringen lykkedes
  C-flaget sat
  Z-flaget resat
Stream'en var i brug
  C-flaget resat
  Z-flaget resat
En fejl opstod
  C-flaget resat
  Z-flaget sat
A, BC, DE, HL, IX og de øvrige flag ødelægges
Bemærk: Rutinen benytter indlæsnings-stream'en, der skal være lukket inden kald af denne rutine. udlæsnings-stream'en berøres overhovedet ikke. De katalogiserede blokke indlæses i den i DE angivne buffer, og startmeddelelser (se evt. CAS NOISY) tillades. Udskrivningen af en blok ser således ud:

  FILNAVN blok X Y Ok

Filnavnet betegnes »Unnamed file«, hvis navnet starter med 00h. X er blokkens nummer, hvor 1 er den første. Y er en angivelse af filtypen. Byte 18 i bufferen angivet under CAS IN OPEN underkastes AND 15 og der tillægges 36. Herved fås flg. muligheder:

$ = et BASIC-program
% = et beskyttet BASIC-program
* = en ASCII fil
& = en binær fil
' = en beskyttet binær fil

Ok angives, hvis der ikke opstod fejl. Katalogiseringen fortsætter, indtil brugeren trykker ESC.

CAS WRITE (20) – BC9Eh – 48286
Laver en optagelse på kassettebåndet

FØR KALD:
HL skal indeholde adressen på de data, der skal udlæses
DE skal indeholde antallet af data (bytes)
A skal indeholde den såkaldte »synkrone karakter«
VED RETURNERING:
Hvis optagelsen er lykkedes
  C-flaget sat
  A ødelægges
Hvis en fejl er opstået eller brugeren har trykket ESC
  C-flaget resat
  A indeholder en fejlkode
BC, DE, HL og IX ødelægges
Bemærk: Denne rutine benyttes af CAS OUT CHAR, CAS OUT DIRECT og CAS OUT CLOSE til at lave optagelser. De data, der skal overføres, må ligge et hvilket som helst sted i RAM, selv under ROM. En længde på 0 angiver en længde på 65536, d.v.s. hele adresseringsområdet. Den såkaldte »synkrone karakter« bruges til at skelne mellem selve de data, der skal udlæses, og de data, der indeholder besked om filen. Sidstnævnte er de under CAS IN OPEN nævnte 64 bytes. Hvis karakteren er 44, er der tale om sidstnævnte type, mens en karakter på 22 angiver data. Rutinen starter selv kassettebåndoptagerens motor, og ved afslutning retableres dens status fra før kald af denne rutine. Fejlkoden, der returneres ved fejl under udlæsningen, kan være

0 = brugeren har trykket ESC
1 = båndoptageren var ikke i stand til at overføre data så hurtigt

CAS READ (21) – BCA1h – 48289
Indlæser en optagelse fra kassettebåndet
FØR KALD:
HL skal indeholde adressen, hvor de indlæste data skal placeres
DE skal indeholde antallet af indlæste data (bytes)
A skal indeholde den forventede »synkrone karakter«
VED RETURNERING:
Hvis indlæsningen af optagelsen er lykkedes
  C-flaget sat
  A ødelægges

Hvis en fejl er opstået eller brugeren har trykket ESC
C-flaget resat
A indeholder en fejlkode
BC, DE, HL, IX og øvrige flag ødelægges
Bemærk: Denne rutine benyttes bl.a. af CAS IN CHAR og CAS IN DIRECT til indlæsning af optagelser fra kassettebånd. Kassettebåndoptagerens motor startes automatisk, og ved indlæsningens afslutning genetableres dens status fra før kald af denne rutine. Hvis længden angives til 0, menes 65536 bytes. Hvis den i DE angivne længde af optagelsen er mindre end den korrekte, bliver kun denne del indlæst. Herved vil hele optagelsen altså ikke blive indlæst. Vedr. den »synkrone karakter«, se CAS WRITE. De mulige fejlkoder er

0 = brugeren har trykket ESC
1 = der er fundet en bit, der er for lang
2 = en fejl ved datablokken

CAS CHECK (22) – BCA4h – 48292
Sammenligner en optagelse på kassettebånd med indholdet i hukommelsen
FØR KALD:
HL skal indeholde adressen på de data, der skal checkes
DE skal indeholde antallet af disse data
A skal indeholde den forventede »synkrone karakter«
VED RETURNERING:
Hvis sammenligningen er lykkedes
C-flaget sat
A ødelægges
Hvis en fejl er opstået eller brugeren har trykket ESC
C-flaget resat
A indeholder en fejlkode
BC, DE, HL, IX og øvrige flag ødelægges
Bemærk: Det er ikke nødvendigt at sammenligne en hel optagelse. Hvis den angivne længde er mindre end antallet af data i optagelsen, sammenlignes blot det opgivne antal bytes. En længde på 0 angiver en længde på 65536 bytes. Kassettebåndoptagerens motor startes af denne rutine, og ved returnering genetableres dens status fra før kaldet af denne rutine. Vedr. den »synkrone

karakter«, se CAS WRITE. Området, der skal sammenlignes, skal ligge i RAM, et hvilket som helst sted, gerne under ROM. De mulige fejlkoder er

0 = brugeren har trykket ESC
1 = båndoptageren ville indlæse en bit, der var for lang
2 = en fejl ved datablokken
3 = sammenligningen faldt uheldigt ud (der var ikke overensstemmelse mellem de i området og de på optagelsen fundne bytes)

GRUPPE 6: LYDEN

SOUND RESET (1) – BCA7h – 48295
Resætter lyddelen i LROM
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Rutinen tømmer alle lydkøer og stopper igangværende lyde.

SOUND QUEUE (2) – BCAAh – 48298
Lægger en lyd i en lydkø
FØR KALD:
HL skal indeholde adressen på en lydblok
VED RETURNERING:
C-flaget sat, hvis lyden er lagt i lydkøen/-køerne
  HL ødelægges
C-flaget resat, hvis mindst en lydkø var fuld
  HL bevares
A, BC, DE, IX og øvrige flag ødelægges
Bemærk: Lydblokken skal ligge i de midterste, centrale 32 K af RAM. Den skal have flg. udseende:

byte 0:	kanalstatus
byte 1:	lydstyrkeændring (volume envelope)
byte 2:	toneændring (tone envelope)
byte 3-4:	tone
byte 5:	støjperiode
byte 6:	lydstyrke v/start
byte 7-8:	tonens varighed

Kanalstatus skal sættes således:

bit 0:	lyd til kanal A
bit 1:	lyd til kanal B
bit 2:	lyd til kanal C
bit 3:	rendezvous med kanal A
bit 4:	rendezvous med kanal B
bit 5:	rendezvous med kanal C
bit 6:	standser lyd
bit 7:	tømmer lydkø

Bit 6 sat medfører, at lyden ikke udsendes, før den frigøres med SOUND RELEASE. Bit 7 sat medfører udover tømningen af lydkøen, at alle nuværende lyde stoppes, men fremtidige tillades.

Lydstyrkeændringen skal ligge inden for 0-15, hvor alle i intervallet 1-15 kan sættes med SOUND AMPL ENVELOPE. 0 angiver ingen lydstyrkeændring, men konstant lydstyrke i henhold til lydrutinens byte 6.

Toneændringen skal ligge inden for 0-15, hvor alle i intervallet 1-15 kan sættes med SOUND TONE ENVELOPE. 0 angiver ingen toneændring, men konstant tone som angivet i lydrutinens byte 3-4.

Tonen skal angives inden for 1-4095, hvor 1 enkelt angiver 8 mikrosekunder. En angivelse på 0 medfører ingen tone overhovedet.

Støjperioden skal ligge inden for 0-31 og angiver, hvor meget støj, der skal tilføjes den enkelte tone. 0 medfører ingen støj.

Lydstyrken skal ligge i intervallet 0-15, hvor 15 er den maksimale lydstyrke.

For tonens varighed gælder specielle regler. Hvis tallet er posi-

tivt (1-32767), angiver det tonens varighed i hundrededele af sekunder. Hvis tallet er nul, vil volume envelope udføres en gang, mens det ved negativt tal (32768-65535) vil udføres det antal gange, det negative tal angiver. Hvis tonens varighed angives kortere end ved lydstyrkeændring, stoppes sidstnævnte efter den angivne varighed. Hvis varigheden af en tone angives længere end lydstyrkeændring, forlænges denne. På samme måde med toneændringen.

På en kø kan kun stå fem lyde; nemlig den der udføres og fire, der venter. Lyde, der herefter forsøges lagt i en kø, vil blive afvist. Hver kanal har sin egen kø.

SOUND CHECK (3) – BCADh – 48301
Undersøger, om der er plads i en lydkø
FØR KALD:
A skal indeholde en værdi, der angiver hvilken kanal, der skal testes
VED RETURNERING:
A indeholder kanalstatus
BC, DE, HL og flagene ødelægges
Bemærk: Værdien, der skal angives i A ved kaldet, fortæller, hvilken kanal der skal testes
bit 0 sat: kanal A
bit 1 sat: kanal B
bit 2 sat: kanal C
Kun én kanal kan testes. Hvis flere bit sættes, vil kun den første af kanalerne blive undersøgt. Kanalstatus ved returneringen er opbygget således:
bit 0-2: frie pladser i lydkøen for den valgte kanal
bit 3 sat: kanalen afventer rendezvous med kanal A
bit 4 sat: kanalen afventer rendezvous med kanal B
bit 5 sat: kanalen afventer rendezvous med kanal C
bit 6 sat: kanalen tilbageholder en lyd
bit 7 sat: kanalen er aktiv
Denne rutine frakobler muligheden for et lyd event, når der er en fri plads i lydkøen, se evt. SOUND ARM EVENT.

SOUND ARM EVENT (4) – BCB0h – 48304
Opretter lyd event, når der er plads i en lydkø

FØR KALD:
A skal indeholde en værdi, der angiver hvilken kanal, der benyttes
HL skal indeholde adressen på en eventblok
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Valget af kanal gøres således:
bit 0 sat: kanal A
bit 1 sat: kanal B
bit 2 sat: kanal C
Kun én kanal kan vælges. Hvis flere bit sættes, vil kun den første af kanalerne blive udvalgt. Event'et kaldes, når der er en fri plads i lydkøen. Denne eventblok skal initialiseres af KL INIT EVENT. Muligheden for kaldet af dette event fjernes med kald af SOUND QUEUE eller SOUND CHECK eller når et event er under udførelse. Eventrutinen skal således kalde denne rutine igen, hvis event'et atter skal kaldes, når der er en fri plads i køen.

SOUND RELEASE (5) – BCB3h – 48307
Frigiver en tilbageholdt lyd
FØR KALD:
A skal indeholde en værdi, der angiver hvilke kanaler, der frigives
VED RETURNERING:
AF, BC, DE, HL og IX ødelægges
Bemærk: Værdien, der angives i A ved kald, skal opbygges således:
bit 0 sat: kanal A frigives
bit 1 sat: kanal B frigives
bit 2 sat: kanal C frigives
En lyd kan tilbageholdes ved hjælp af en sat bit ved kald af SOUND QUEUE (rutine 2 i denne gruppe).

SOUND HOLD (6) – BCB6h – 48310
Stopper alle lyde midlertidigt
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis en lyd var under udførelse

C-flaget resat, hvis ingen lyd var under udførelse
A, BC, HL og øvrige flag ødelægges
Bemærk: Lydene kan frigives med SOUND CONTINUE, eller med SOUND QUEUE eller SOUND RELEASE. Førstnævnte vil være mest nærliggende.

SOUND CONTINUE (7) – BCB9h – 48313
Igangsætter alle midlertidigt tilbageholdte lyde
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og IX ødelægges
Bemærk: Lydene, der frigives, er dem, der er stoppet af SOUND HOLD. Hvis ingen lyde er tilbageholdt, sker der intet.

SOUND AMPL ENVELOPE (8) – BCBCh – 48316
Opretter en lydstyrkeændring (volume envelope)
FØR KALD:
A skal indeholde lydstyrkeændringens (envelope'ns) nummer
HL skal indeholde en adresse på en datablok
VED RETURNERING:
C-flaget sat, hvis oprettelsen er lykkedes
  HL indeholder adressen på datablokken tillagt 16
  A og BC ødelægges
C-flaget resat, hvis nummeret på lydstyrkeændringen er ugyldigt
  A, BC og HL bevares
DE og øvrige flag ødelægges
Bemærk: Lydstyrkeændringens nummer (det såkaldte envelope nummer) skal ligge inden for intervallet 1-15. Datablokken skal ligge enten i RAM eller i ROM, men i RAM må den ikke ligge under ROM. En datablok består af op til 16 bytes. Hver envelope kan bestå af op til fem sektioner.
Datablokkens udseende er derfor:
byte 0 : antallet af sektioner i envelope'n
byte 1- 3: 1. sektion
byte 4- 6: 2. sektion
byte 7- 9: 3. sektion
byte 10-12: 4. sektion
byte 13-15: 5. sektion

Til hver sektion hører således tre bytes. Antallet af sektioner skal ligge inden for 1-5. Angives 0, holdes en konstant lydstyrke i to sekunder. Andre angivelser kan medføre uforudsigelige hændelser.

Der findes både såkaldte »software envelope« og »hardware envelope«. Her nævnes kun den førstnævnte, der indikeres ved en resat bit 7 i den første byte i den enkelte sektion. En sektion er opbygget således:

byte 0: skridttæller (antallet af skridt)
byte 1: skridtstørrelse
byte 2: intervallængde (tiden mellem hvert skridt)

Skridttælleren skal angives i intervallet 1-127 (bit 7's betydning er forklaret), og det angiver antallet af skridt. En skridttæller på 0 angiver konstant lydstyrke. Der er intervallængden, der angives i hundrededele sekunder, mellem hvert skridt, og selve skridtstørrelsen er angivet i byte 1. Den udregnede lydstyrke vil aldrig overstige 15. En intervallængde på 0 angiver 256.

»Hardware envelope« forklares ikke.

SOUND TONE ENVELOPE (9) – BCBFh – 48319

Producerer en toneændring (tone envelope)

FØR KALD:

A skal indeholde toneændringens nummer
HL skal indeholde en adresse på en datablok

VED RETURNERING:

C-flaget sat, hvis oprettelsen er lykkedes
  HL indeholder adressen på datablokken tillagt 16
  A og BC ødelægges
C-flaget resat, hvis nummeret på toneændringen er ugyldigt
  A, BC og HL bevares
DE og øvrige flag ødelægges

Bemærk: Toneændringens nummer (det såkaldte envelope nummer) skal ligge i intervallet 1-15. Datablokken skal ligge i ROM eller RAM, i sidstnævnte dog ikke under ROM. Datablokkens udseende kan iagttages under SOUND AMPL ENVELOPE. Den består af 16 bytes, og et envelope kan bestå af op til fem sektioner, der hver består af tre bytes. Antallet af sektioner er 1-5. En angivelse på 0 medfører en konstant tone. Andre angivelser kan få uforudsigelige konsekvenser. En sektion er opbygget som en sek-

tion i SOUND AMPL ENVELOPE. Hvis bit 7 i skridttælleren er sat, vil envelope'n ved dens afslutning begynde forfra. Hvis skridttælleren ligger i intervallet 0-239, vil tonen blive ændret det antal gange, som skridttælleren angiver. Ændringen sker med skridtstørrelsen, og der er intervallængden, angivet i hundrededele sekunder, mellem hvert skridt. En intervallængde på 0 betyder 256. En skridttæller på 0 betyder 1 skridt. Hvis skridttælleren er i intervallet 240-255 beregnes en tone, der holdes i den angivne intervallængde.

SOUND A ADDRESS (10) – BCC2h – 48322
Finder adressen på en lydstyrkeændring (volume envelope)
FØR KALD:
A skal indeholde nummeret på lydstyrkeændringen
VED RETURNERING:
C-flaget sat, hvis lydstyrkeændringen er fundet
  HL indeholder adressen på lydstyrkeændringen
  BC indeholder længden af dens datablok (16 bytes)
C-flaget resat, hvis lydstyrkeændringen ikke er fundet
  HL ødelægges
  BC bevares
A og øvrige flag ødelægges
Bemærk: Nummeret på lydstyrkeændringen skal ligge i intervallet 1-15. Se evt. SOUND AMPL ENVELOPE.

SOUND T ADDRESS (11) – BCC5h – 48325
Finder adressen på en toneændring (tone envelope)
FØR KALD:
A skal indeholde nummeret på toneændringen
VED RETURNERING:
C-flaget sat, hvis toneændringen er fundet
  HL indeholder adressen på toneændringen
  BC indeholder længden af dens datablok (16 bytes)
C-flaget resat, hvis toneændringen ikke er fundet
  HL ødelægges
  BC bevares
A og øvrige flag ødelægges
Bemærk: Nummeret på toneændringen skal ligge i intervallet 1-15. Se evt. SOUND TONE ENVELOPE.

GRUPPE 7: THE KERNEL

KL CHOKE OFF (1) – BCC8h – 48328
Resætter kernen
FØR KALD:
Ingen krav
VED RETURNERING:
B indeholder nummeret på den benyttede forgrunds-ROM
DE indeholder adressen på den benyttede forgrunds-ROM
C indeholder nummeret på en forgrunds-RAM
AF og HL ødelægges
Bemærk: Der findes to typer ROM, nemlig forgrunds- og baggrunds-ROM. Desuden findes en systemudvidelse (RSX), men den skal ligge i RAM. Et eksempel på en forgrunds-ROM er den indbyggede BASIC. Andre eksempler er en assembler eller andre sprog. Baggrunds-ROM beskæftiger sig med det tilsluttede ekstraudstyr. Uden nogen udvidelse anvendes ROM nr. 0 af de 252 mulige, som systemet kan udvides til.

Denne rutine benyttes ved indlæsning af programmer, hvor alle events og interrupts skal passiviseres. Den tømmer alle eventkøer m.m.

KL ROM WALK (2) – BCCBh – 48331
Initialiserer alle baggrunds-ROM
FØR KALD:
DE skal indeholde adressen på den først brugbare byte
HL skal indeholde adressen på den sidst brugbare byte
VED RETURNERING:
DE indeholder den nye, først brugbare adresse
HL indeholder den nye, sidst brugbare adresse
AF og BC ødelægges
Bemærk: Når et program skal indlæses i hukommelsen, skal der først gøres plads til alle baggrunds-ROM. Denne rutine finder og initialiserer alle disse, så de er anvendelige til brug ved udvidelsesudstyr, se evt. KL CHOKE OFF og KL INIT BACK.

KL INIT BACK (3) – BCCEh – 48334
Initialiserer én baggrunds-ROM

FØR KALD:
C skal indeholde nummeret på den ROM, der skal initialiseres
DE skal indeholde adressen på den først brugbare byte
HL skal indeholde adressen på den sidst brugbare byte
VED RETURNERING:
DE indeholder den nye, først brugbare adresse
HL indeholder den nye, sidst brugbare adresse
AF og B ødelægges
Bemærk: ROM-nummeret skal ligge mellem 1 og 7. Den først brugbare byte ligger adressemæssigt nederst i hukommelsen, mens den sidst brugbare ligger øverst. Når et program skal indlæses i hukommelsen, skal baggrunds-ROM initialiseres, så disse kan benyttes. Denne rutine finder én specielt baggrunds-ROM og initialiserer denne. For brugen af baggrunds-ROM, se evt. KL CHOKE OFF.

KL LOG EXT (4) – BCD1h – 48337
Producerer en systemudvidelse (RSX)
FØR KALD:
BC skal indeholde adressen på en kommandotabel
HL skal indeholde adressen på et område indeholdende 4 bytes til brug for systemet
VED RETURNERING:
DE ødelægges
Bemærk: En systemudvidelse består af en eller flere kommandoer, som tilføjes det benyttede system. Kommandoerne tilføjes listen af disse under betegnelsen »eksterne kommandoer«. Udvidelsen skal være beliggende i de midterste, centrale 32 K af hukommelsen (RAM), og må ikke være beliggende i ROM. En RSX er i øvrigt mage til en baggrunds-ROM, bortset fra beliggenheden. Kommandotabellen, hvis startadresse skal angives i BC, skal have flg. udseende (eksemplet fra kapitel 23 er benyttet):

```
DEFB NAVNETABEL (adressen på navnetabellen)
JP NED (NED er navnet på den første kommando)
JP OP (navnet på den anden kommando)
NAVNETABEL
DEFB N, E, (D + 128) (kommandonavnets bogstaver)
DEFB 0, (P + 128) (det andet kommandonavns bogstaver)
DEFB 0
```

Som det ses angives først i 2 bytes adressen på den navnetabel, som følger senere i kommandotabellen. Dernæst følger tre bytes for hver kommando. Den første af disse tre er C3h (195 – JP nn), og de to næste adressen, hvor den rutine, der skal kaldes når kommandoen benyttes, står. I eksemplet er således til dette ialt anvendt 6 bytes. Herefter følger navnetabellen, hvor der blot står alle bogstaverne i kommandonavnene. Navnene må højst være på 16 karakterer, og alle skal have koder inden for intervallet 0-127. Den 1. karakter må ikke være 0. For den indbyggede BASIC kan der opstå problemer ved brug af f.eks. komma, så dette bør undgås. Det sidste bogstav i et navn skal have bit 7 sat. I ovenstående eksempel fylder navnene således 3 + 2 = 5 bytes. Hele kommandotabellen afsluttes med en byte indeholdende 0.

KL FIND COMMAND (5) – BCD4h – 48340
Leder efter en kommando
FØR KALD:
HL skal indeholde adressen på det kommandonavn, som skal eftersøges
VED RETURNERING:
C-flaget sat, hvis kommandoen er fundet
  C indeholder ROM-nummeret
  HL indeholder adressen på rutinen
C-flaget resat, hvis kommandoen ikke er fundet
  C og HL ødelægges
A, B og DE ødelægges
Bemærk: Kommandonavnet skal ligge i RAM, gerne under ROM. Kun de første 16 karakterer af dets navn kontrolleres, hvorved overskydende tegn er uden betydning. Den sidste karakter skal have bit 7 sat.

Rutinen gennemsøger på listen over »eksterne kommandoer« fra RSX og baggrunds-ROM for at finde den kommando, der søges. Hvis kommandoen ikke findes, undersøges forgrunds-ROM (den indbyggede BASIC).

Hvis kommandoen findes, returneres ROM-nummer og adresse. Dette kan benyttes til rutinen KL FAR PCHL i »The Low Kernel Jumpblock«.

KL NEW FRAME FLY (6) – BCD7h – 48343
Initialiserer og lægger en eventblok på listen over events, der skal udføres ved hvert frame flyback interrupt
FØR KALD:
B skal indeholde eventklassen
C skal indeholde ROM-nummeret
DE skal indeholde adressen på eventrutinen
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, DE og HL ødelægges
Bemærk: Eventblokken er 9 bytes lang og skal ligge i de midterste 32 K af RAM. Dens opbygning er således:
byte 0-1: systembytes til brug ved frame flyback interrupt
byte 2-8: normal eventblok, se KL INIT EVENT

De første to bytes er kun benyttet ved et frame flyback interrupt, mens de syv øvrige er den normale eventblok. Rutinen initialiserer et event, der vil blive lagt på listen over de, der skal udføres hver gang et frame flyback interrupt kommer. Dette sker 50 gange i sekundet. For generel orientering om events, se KL INIT EVENT.

KL ADD FRAME FLY (7) – BCDAh – 48346
Tilfører et event på listen over de, der skal udføres hver gang et frame flyback interrupt kommer
FØR KALD:
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, DE og HL ødelægges
Bemærk: Rutinen tilføjer et event til listen over de, der skal udføres hver gang et frame flyback interrupt kommer. Eventblokken, som beskriver det pågældende event, skal være initialiseret i forvejen. Se KL INIT EVENT for en generel orientering om events.

KL DEL FRAME FLY (8) – BCDDh – 48349
Fjerner et event fra listen over de, der skal udføres hver gang et frame flyback interrupt kommer
FØR KALD:
HL skal indeholde adressen på eventblokken

VED RETURNERING:
AF, DE og HL ødelægges
Bemærk: Hvis det pågældende event ikke er på listen, vil der intet ske. For en generel orientering om events, se KL INIT EVENT.

KL NEW FAST TICKER (9) – BCE0h – 48352
Initialiserer og lægger et event på listen over de, der skal udføres hver gang et fast ticker interrupt kommer
FØR KALD:
B skal indeholde eventklassen
C skal indeholde ROM-nummeret
DE skal indeholde startadressen på eventrutinen
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, DE og HL ødelægges
Bemærk: Denne rutine både initialiserer en eventblok og lægger dette event på listen over de, der skal udføres, når et fast ticker interrupt kommer. Dette sker 300 gange i sekundet, hvorfor det kan være nyttigt til måling af korte tidsrum.

Eventblokken har følgende udseende:
byte 0-1: systembytes til brug ved fast ticker interrupt
byte 2-8: normal eventblok, se KL INIT EVENT

Blokken er således på 9 bytes, hvoraf de to første kun benyttes ved et fast ticker interrupt. Se KL INIT EVENT for en generel orientering om events.

KL ADD FAST TICKER (10) – BCE3h – 48355
Tilføjer et event til listen over de, der skal udføres hver gang et fast ticker interrupt kommer
FØR KALD:
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, DE og HL ødelægges
Bemærk: Rutinen tilføjer det pågældende event til listen over de, der skal udføres ved ethvert fast ticker interrupt. Inden kald af denne rutine skal eventblokken være initialiseret. Se KL NEW FAST TICKER for oplysning om dette interrupt og KL INIT EVENT for en generel orientering om events.

KL DEL FAST TICKER (11) – BCE6h – 48358
Fjerner et event fra listen over de, der skal udføres ved hvert fast ticker interrupt
FØR KALD:
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, DE og HL ødelægges
Bemærk: Hvis det pågældende event ikke er på listen, vil der intet ske. Se KL INIT EVENT for en generel orientering om events og KL NEW FAST TICKER for oplysningen om fast ticker interrupt.

KL ADD TICKER (12) – BCE9h – 48361
Tilføjer et event til listen over de, der kan blive udført ved et ticker interrupt
FØR KALD:
HL skal indeholde adressen på eventblokken
DE skal indeholde startværdien for tæller
BC skal indeholde tællerens startværdi efter hver udførelse af et event
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Det pågældende event vil ikke blive udført ved ethvert ticker interrupt, der kommer 50 gange i sekundet. Det afhænger af en indbygget tæller, der decrementeres ved hvert interrupt. Når tælleren når nul, udføres det pågældende event og tælleren initialiseres påny. Eventblokken har følgende udseende:
byte 0- 1: systembytes til brug ved ticker interrupt
byte 2- 3: tæller
byte 4- 5: tællerens startværdi efter hver udførelse
byte 6-12: normal eventblok, se KL INIT EVENT

Blokken er således på 13 bytes, hvoraf rutinen initialiserer de 6 første. De 7 sidste svarer til en normal eventblok, se evt. KL INIT EVENT, og den skal være initialiseret inden kald af denne rutine. Hvis startværdien for tælleren angives til 0, ingnoreres hele eventblokken og dermed muligheden for udførelse af et event ved hvert ticker interrupt. Hvis tællerens startværdi efter hvert events udførelse sættes til 0, vil event'et kun forekomme én gang, hvorefter det passiviseres. Se KL INIT EVENT for en generel orientering om events.

KL DEL TICKER (13) – BCECh – 48364
Fjerner et event fra listen over de, der kan blive udført ved hvert ticker interrupt
FØR KALD:
HL skal indeholde adressen på eventblokken
VED RETURNERING:
C-flaget sat, hvis det pågældende event blev fundet på listen
DE indeholder tællerens øjeblikkelige værdi
C-flaget resat, hvis det pågældende event ikke blev fundet på listen
DE ødelægges
A, HL og øvrige flag ødelægges
Bemærk: Eventblokkens indhold påvirkes ikke af, at det pågældende event fjernes fra listen. Hvis eventblokken således atter placeres på listen, kan den genoptage sin tælling fra det sted, den slap. Se KL ADD TICKER for en uddybning af events ved ticker interrupt og KL INIT EVENT for en generel orientering om events.

KL INIT EVENT (14) – BCEFh – 48367
Initialiserer en eventblok
FØR KALD:
B skal indeholde eventklassen
C skal indeholde ROM-nummeret
DE skal indeholde eventrutinens startadresse
HL skal indeholde adressen på eventblokken
VED RETURNERING:
HL indeholder adressen på eventblokken tillagt 7
Bemærk: Eventblokken er 7 bytes lang og skal ligge i de midterste 32 K af RAM. Den ser således ud:
byte 0-1: systembytes til brug ved alle events
byte 2 : tæller (event count)
byte 3 : eventklasse
byte 4-5: eventrutinens startadresse
byte 6 : ROM-nummer
byte 7 m.fl.: brugerområde

Benyttelsen af brugerområdet, der ligger udover de 7 bytes, er frivillig. De to første bytes bruges af systemet, og brugeren må ikke benytte dem. Ved både frame flyback interrupt, fast ticker

interrupt og ticker interrupt benyttes yderligere 2 bytes til systemet. Se de respektive rutiner for dette.

Tælleren benyttes til at tælle antallet af events, der er genereret, men ikke udført. Eventklassen fortæller om typen af det pågældende event, og værdien findes således:

bit 0 : sat – 2 bytes-adresse
resat – 3 bytes-adresse
bit 1-4: prioriteten af synkroniske event
bit 5 : skal være resat
bit 6 : sat – ekspress event
resat – normal event
bit 7 : sat – asynkron event
resat – synkron event

Der er to typer events: asynkrone og synkrone. De synkrone udføres først, når det findes belejligt af brugeren. Ekspress synkrone events har højere prioritet end normal synkrone events, og de udføres derfor før. Prioriteten kan sættes med de fire angivne bit. Jo højere nummer, desto højere prioritet. Asynkrone events udføres altid. Ekspress asynkrone events udføres direkte ved det indkomne interrupt, mens normal asynkrone events først udføres, når der returneres fra interruptrutinen. Desuden angives for alle events en adresse, enten på 2 eller 3 bytes. Hvis bit 0 i eventklassen er sat, er der tale om en 2 bytes-adresse, og der hoppes direkte til eventrutinen. Er der tale om en 3 bytes-adresse, angives desuden et ROM-nummer. Hvis der bruges en 2 bytes-adresse, hvor ROMs nummer er ligegyldigt, ignoreres C-registrets værdi ved kaldet.

Asynkrone events har, da de udføres ved interrupt'et, ingen prioritet.

Eventblokken, der beskriver det pågældende event, initialiseres således med denne rutine. Brugeren behøver kun tage sig af eventklassen (og ROM-nummeret), eventblokkens adresse og eventrutinens adresse. Resten ordner systemet selv. Eventrutinen er den rutine, der udføres, når et event er klart til udførelse. Med den følger disse karakteristika:

FØR EVENTRUTINEN BRUGES:

HL skal indeholde adressen på byte 5 i eventblokken ved en 3 bytes-adresse

HL skal indeholde adressen på byte 6 i eventblokken ved en 2 bytes-adresse
NÅR EVENTRUTINEN RETURNERER:
AF, BC, DE og HL ødelægges
Desuden skal eventrutinen gemme IX og IY's værdi, og det alternative registersæt må ikke benyttes.

Desuden gælder, at ekspress asynkrone events ikke må tillade interrupts og eventrutinens adresse skal angives som en 2 bytes-adresse.

KL EVENT (15) – BCF2h – 48370
Genererer et event
FØR KALD:
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Der genereres et event manuelt. Hvis tælleren er positiv (eller 0), genereres det pågældende event, hvorimod en negativ tæller medfører, at der intet sker. Synkrone events, der er genereret, lægges på køen og udføres, når rutinen KL DO SYNC kaldes. Asynkrone events udføres næsten med det samme, afhængig af dets type. For en fuldstændig forklaring af de to typer, se KL INIT EVENT. Tælleren incrementeres ved hver generering og decrementeres ved hver udførelse.

KL SYNC RESET (16) – BCF5h – 48373
Rydder køen af synkrone events
FØR KALD:
Ingen krav
VED RETURNERING:
AF og HL ødelægges
Bemærk: Alle events på den kø, der er forbeholdt synkrone events, fjernes.

KL DEL SYNCHRONOUS (17) – BCF8h – 48376
Fjerner et synkront event fra eventkøen
FØR KALD:
HL skal indeholde adressen på eventblokken

VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Før en ændring i en eventblok bør denne rutine kaldes for at sikre mod uhensigtsmæssige fejl. Det på gældende event gøres passivt og fjernes desuden fra eventkøen, hvis det står derpå.

KL NEXT SYNC (18) – BCFBh – 48379
Henter et event fra køen til udførelse
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis der var et event på køen klar til udførelse
  HL indeholder adressen på det pågældende events eventblok
  A indeholder det foregående events prioritet
C-flaget resat, hvis der intet event var på køen
  A og HL ødelægges
DE ødelægges
Bemærk: Hvis der er et event på køen, der har større prioritet end det nuværende, hentes dette fra køen, og det nuværende returneres. Hvis der intet nuværende event er, hentes event'et under alle omstændigheder. Udførelsen af det synkrone event fra køen bør foregå således:
HOVEDPROGRAM:
CALL KL POLL SYNCHRONOUS
CALL C, UDFØRELSESRUTINE
UDFØRELSESRUTINE:
CALL KL NEXT SYNC
RET NC
PUSH HL
PUSH AF
CALL KL DO SYNC
POP AF
POP HL
CALL KL DONE SYNC
JR - 16
Udførelsesrutinen på 16 bytes kaldes således kun, hvis det er nødvendigt. Denne rutine returnerer prioriteten af det foregående

event, og det skal overføres til KL DONE SYNC, hvilket ses i udførelsesrutinen.

KL DO SYNC (19) – BCFEh – 48382
Udfører en eventrutine
FØR KALD:
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Denne rutine udfører det event, som KL NEXT SYNC har fundet skulle udføres. Se KL NEXT SYNC for den rette sammenhæng for denne rutine. Anden brug end den der angivne kan ikke anbefales.

KL DONE SYNC (20) – BD01h – 48385
Færdigudfører et event
FØR KALD:
A skal indeholde det forrige events prioritet
HL skal indeholde adressen på eventblokken
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: Denne rutine skal kaldes efter KL DO SYNC. Se KL NEXT SYNC for denne rutines rette sammenhæng. Den benyttes til at etablere den rigtige prioritet og til decrementering af tælleren (event count). Hvis den forbliver større end nul, placeres event'et på køen igen.

KL EVENT DISABLE (21) – BD04h – 48388
Frakobler muligheden for udførelse af normal synkroniske events
FØR KALD:
Ingen krav
VED RETURNERING:
HL ødelægges
Bemærk: Alle normale synkrone events forhindres udført, idet prioriteten af det nuværende event sættes højere end ethvert normal synkronisk event. Ekspress synkrone events tillades derimod. Genereringen af normal synkrone events forhindres således ikke – de udføres blot ikke.

KL EVENT ENABLE (22) – BD07h – 48391
Tillader udførelse af normale synkrone events
FØR KALD:
Ingen krav
VED RETURNERING:
HL ødelægges
Bemærk: Rutinen tillader udførelse af normale synkrone events og ophæver dermed virkningen af KL EVENT DISABLE.

KL DISARM EVENT (23) – BD0Ah – 48394
Forhindrer generering af et event
FØR KALD:
HL skal indeholde adressen på en eventblok
VED RETURNERING:
AF ødelægges
Bemærk: Det pågældende event kan ikke genereres, da tælleren (event count) gøres negativ. Det bør kun benyttes til asynkrone events, da de synkrone bør forhindres med KL DEL SYNCHRONOUS (rutine 17 i denne gruppe). Genereringen kan tillades igen enten ved at sætte tælleren i eventblokken direkte eller ved en reinitialisering af eventblokken med KL INIT EVENT.

KL TIME PLEASE (24) – BC0Dh – 48397
Undersøger tiden i det indbyggede ur
FØR KALD:
Ingen krav
VED RETURNERING:
DEHL indeholder tællerens værdi
Bemærk: Tælleren er en fire bytes-tæller, hvor D er MSB og L LSB. Tælleren incrementeres for hvert time interrupt, der kommer 300 gange i sekundet. Tælleren nulstilles ved start af computeren og ved reset. Den kan sættes med KL TIME SET.

KL TIME SET (25) – BD10h – 48400
Sætter tiden i det indbyggede ur
FØR KALD:
DEHL skal indeholde den ønskede tid
VED RETURNERING:
AF ødelægges

Bemærk: Tiden i det indbyggede ur er en 4 bytes-tæller, hvor D er MSB og L LSB. Tælleren incrementeres 300 gange i sekundet og kan aflæses med KL TIME PLEASE.

GRUPPE 8: MACHINE PACK

MC BOOT PROGRAM (1) – BD13h – 48403
Indlæser og starter et program
FØR KALD:
HL skal indeholde adressen på den rutine, der indlæser programmet
VED RETURNERING:
Returnerer ikke!
Bemærk: Rutinen bruges til at indlæse og starte et RAM-program. Inden indlæsningen kaldes følgende:
SOUND RESET (alle lyde stoppes, lydkøer tømmes)
KL CHOKE OFF (alle eventkøer tømmes)
KM RESET (tastaturet resættes)
TXT RESET (tekstskærmen resættes)
SCR RESET (skærmen resættes)
Desuden forhindres eksterne interrupts og den oprindelige stak retableres m.m. Rutinen, der indlæser programmet, skal ligge i en læsbar del af RAM eller en tilkoblet ROM. Normalt tilkobles UROM, og LROM frakobles, hvorfor programmet skal ligge enten i UROM eller i RAM fra adresse 16384 og opefter. Rutinen, hvis startadresse HL peger på, bør returnere med følgende værdier:
C-flaget sat, hvis indlæsningen er lykkedes
  HL indeholder programmets startadresse
C-flaget resat, hvis indlæsningen mislykkedes
  HL ødelægges
A, BC, DE, IX, IY og øvrige flag ødelægges
  Hvis indlæsningen mislykkes, skal det foregående program bibeholdes. Hvis indlæsningen lykkes, initialiseres operativsystemet og programmet starter ved den i HL returnerede adresse.

MC START PROGRAM (2) – BD16h – 48406
Starter et program
FØR KALD:
HL skal indeholde programmets startadresse
C skal indeholde ROM-nummeret
VED RETURNERING:
Returnerer ikke!
Bemærk: Rutinen benyttes til indlæsning af forgrunds-ROM og til start af allerede indlæste programmer i RAM. Rutinen omfatter en total initialisering af computeren som ved start.

MC WAIT FLYBACK (3) – BD19h – 48409
Venter på det næste frame flyback interrupt
FØR KALD:
Ingen krav
VED RETURNERING:
Alt bevares
Bemærk: Rutinen venter på det næste frame flyback, af hvilke der kommer 50 i sekundet. I venteperioden ajourføres skærmen ikke, hvorved rutinen kan blive nyttig ved grafik og brug af farver.

MC SET MODE (4) – BD1Ch – 48412
Sætter skærmens MODE
FØR KALD:
A skal indeholde den ønskede MODE
VED RETURNERING:
AF ødelægges
Bemærk: Rutinen fortæller maskinellet, hvilken skærmtilstand, der benyttes. Tallet i A skal ligge i intervallet 0-2, men det kontrolleres, og der sker intet, hvis det er ulovligt. Normalt skal SCR SET MODE benyttes til ændring af skærmens MODE, se evt. denne rutine.

MC SCREEN OFFSET (5) – BD1Fh – 48415
Sætter skærmens off-set og dens startadresse
FØR KALD:
A skal indeholde MSB af den første skærmadresse
HL skal indeholde skærmens off-set

VED RETURNERING:
AF ødelægges
Bemærk: Skærmens startadresse legaliseres ved brug af AND 192, og desuden lovliggøres også skærmens off-set, der ikke må overstige 2046. Rutinen fortæller maskinellet om startadressen og off-set. Normalt skal SCR SET BASE og SCR SET OFFSET benyttes til dette brug. Se disse rutiner for yderligere forklaring af de to begreber.

MC CLEAR INKS (6) – BD22h – 48418
Sætter alle inks til én farve
FØR KALD:
DE skal indeholde adressen på et område indeholdende oplysninger om farven
VED RETURNERING:
AF ødelægges
Bemærk: Området angivet ved startadressen i DE skal være på to bytes:
den 1. byte: farven til skærmkanten (BORDER)
den 2. byte: farven til alle inks
Alle 16 inks sættes til den angivne farve. De kan bagefter sættes nøjere med MC SET INKS. Metoden bruges bl.a. til clearing af skærmen.

MC SET INKS (7) – BD25h – 48421
Sætter alle inks
FØR KALD:
DE skal indeholde startadressen på et område indeholdende oplysninger om de 16 inks og skærmkanten
VED RETURNERING:
AF ødelægges
Bemærk: Området skal bestå af 17 bytes af have flg. udseende:
Byte 0: farven til skærmkanten (BORDER)
Byte 1: farven til ink 0
Byte 2: farven til ink 1
Byte 3: farven til ink 2, o.s.v.
Byte 16: farven til ink 15

MC RESET PRINTER (8) – BD28h – 48424
Initialiserer rutinen MC WAIT PRINTER
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges
Bemærk: MC WAIT PRINTER er en rutine i jumpblock »Firmware Indirections« beliggende på BDF1h, der benyttes af MC PRINT CHAR. Rutinen MC WAIT PRINTER afgør, hvorvidt en karakter er blevet sendt til printeren eller ej, og uden ændring i rutinen behøves denne rutine (MC RESET PRINTER) ikke at blive kaldt.

MC PRINT CHAR (9) – BD2Bh – 48427
Prøver at sende en karakter til Centronics-porten
FØR KALD:
A skal indeholde karakteren, der skal sendes
VED RETURNERING:
C-flaget sat, hvis karakteren er sendt
C-flaget resat, hvis karakteren ikke er sendt
A og øvrige flag ødelægges
Bemærk: Bit 7 i karakteren ignoreres. Hvis printeren er optaget for lang tid (ca. 0,4 sek.), vil denne rutine returnere uden karakteren sendt.

MC BUSY PRINTER (10) – BD2Eh – 48430
Afprøver, om Centronics-porten er optaget
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis porten er optaget
C-flaget resat, hvis porten er ledig
Øvrige flag ødelægges

MC SEND PRINTER (11) – BD31h – 48433
Sender en karakter til Centronics-porten
FØR KALD:
A skal indeholde den karakter, der skal sendes

VED RETURNERING:
C-flaget sat
A og øvrige flag ødelægges
Bemærk: Når denne rutine kaldes, må printeren ikke være optaget. MC PRINT CHAR kan foretrækkes i stedet.

MC SOUND REGISTER (12) – BD34h – 48436
Sender data til et af lydprocessorens registre
FØR KALD:
A skal indeholde nummeret på registre
C skal indeholde den information, der skal sendes
VED RETURNERING:
AF og BC ødelægges
Bemærk: Denne rutine programmerer ét af lydprocessorens 16 registre direkte. Dette er ikke umiddelbart tilrådeligt, og rutinerne i gruppe 6 må foreslås.

GRUPPE 9:

JUMP RESTORE (1) – BD37h – 48439
Retablerer hele den normale jumpblock
FØR KALD:
Ingen krav
VED RETURNERING:
AF, BC, DE og HL ødelægges

»THE HIGH KERNEL JUMPBLOCK«

KL U ROM ENABLE (1) – B900h – 47360
Tilkobler UROM
FØR KALD:
Ingen krav

VED RETURNERING:
A indeholder den foregående ROM-status
Flagene ødelægges
Bemærk: Upper ROM beliggende fra 49152-65535 tilkobles, og aflæsning i dette område vil medføre aflæsning i den benyttede, tilkoblede ROM. KL ROM RESTORE kan genindsætte den tidligere status.

KL U ROM DISABLE (2) – B903h – 47363
Frakobler UROM
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder den foregående ROM-status
Flagene ødelægges
Bemærk: Upper ROM er beliggende fra 49152-65535, og den frakobles. Aflæsning i dette område vil medføre aflæsning i RAM. KL ROM RESTORE kan genindsætte den tidligere status.

KL L ROM ENABLE (3) – B906h – 48366
Tilkobler LROM
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder den foregående ROM-status
Flagene ødelægges
Bemærk: Lower ROM beliggende i intervallet 0-16383 tilkobles, hvorved aflæsning i dette område vil medføre aflæsning i den tilkoblede ROM. Den foregående ROM-status kan genindsættes med KL ROM RESTORE. LROM er kun tilkoblet, når der benyttes en rutine i operativsystem (firmware).

KL L ROM DISABLE (4) – B909h – 47369
Frakobler LROM
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder den foregående ROM-status
Flagene ødelægges

Bemærk: Lower ROM, der er beliggende fra 0-16383, frakobles. Aflæsning i dette område vil medføre aflæsning i RAM. KL ROM RESTORE kan genindsætte den tidligere ROM-status.

KL ROM RESTORE (5) – B90Ch – 47372
Genindsætter den tidligere ROM-status
FØR KALD:
A skal indeholde den tidligere ROM-status
VED RETURNERING:
AF ødelægges
Bemærk: Den tidligere ROM-status returneres af:
KL U ROM ENABLE
KL U ROM DISABLE
KL L ROM ENABLE
KL L ROM DISABLE
KL ROM SELECT

KL ROM SELECT (6) – B90Fh – 47375
Vælger en ROM
FØR KALD:
C skal indeholde nummeret på den ønskede ROM
VED RETURNERING:
B indeholder den tidligere ROM-status
C indeholder nummeret på den tidligere benyttede ROM
AF ødelægges
Bemærk: Rutinen vælger en angivet ROM og tilkobler denne. Den tidligere ROM-status kan genindsættes ved hjælp af KL ROM RESTORE, og begge de returnerede værdier kan benyttes i KL ROM DESELECT til at genvælge en ROM.

KL CURR SELECTION (7) – B912h – 47378
Undersøger, hvilken UROM der benyttes
FØR KALD:
Ingen krav
VED RETURNERING:
A indeholder nummeret på den i øjeblikket benyttede ROM
Bemærk: Den returnerede værdi kan benyttes i forbindelse med en 3 bytes-adresse ved et event. Se KL INIT EVENT for en orientering om dette.

KL PROBE ROM (8) – B915h – 47381
Undersøger typen af en givet ROM
FØR KALD:
C skal indeholde nummeret på den ROM, der skal undersøges
VED RETURNERING:
A indeholder ROMs klasse
L indeholder ROMs kendetegn
H indeholder ROMs version
B og flagene ødelægges
Bemærk: De første bytes af en UROM indeholder oplysninger om den. ROMs klasse angives således:
0: Forgrunds-ROM
1: Baggrunds-ROM
2: Udvidelses-forgrunds-ROM
Bit 7 er sat, hvis det er den indbyggede ROM, der testes. Kendetegn og version svarer til den ROM, der benyttes, og der kan intet generelt siges.

KL ROM DESELECT (9) – B918h – 47384
Genvælger en ROM
FØR KALD:
B skal indeholde den foregående ROM-status
C skal indeholde nummeret på den forrige ROM
VED RETURNERING:
C indeholder nummeret på den nuværende valgte ROM
B ødelægges
Bemærk: Denne rutine genvælger den ROM, som KL ROM SELECT fjernede. De i den rutine returnerede værdier kan benyttes her. ROM-status genetableres også.

KL LDIR (10) – B91Bh – 47387
Eksekverer en LDIR-instruktion med LROM og UROM frakoblet
FØR KALD:
BC, DE og HL skal sættes som til LDIR
VED RETURNERING:
BC, DE, HL og flagene indeholder som efter LDIR
Bemærk: Rutinen foretager en LDIR-instruktion, hvor både UROM og LROM er frakoblet. Dette betyder, at ROM-status er underordnet.

KL LDDR (11) – B91Eh – 47390
Eksekverer en LDDR-instruktion med LROM og UROM frakoblet
FØR KALD:
BC, DE og HL skal sættes som til LDDR
VED RETURNERING:
BC, DE, HL og flagene indeholder som efter LDDR
Bemærk: Rutinen foretager en LDDR-instruktion, hvor både UROM og LROM er frakoblet. Dette betyder, at ROM-status er underordnet.

KL POLL SYNCHRONOUS (12) – B921h – 47393
Undersøger, om der venter et event med højere prioritet end det nuværende under udførelse
FØR KALD:
Ingen krav
VED RETURNERING:
C-flaget sat, hvis der venter et event med højere prioritet
C-flaget resat, hvis der ikke venter et event med højere prioritet
A og øvrige flag ødelægges
Bemærk: Denne rutine sammenligner det nuværende event's prioritet med det, der venter på køens. Hvis køen til synkrone events er tom, returneres øjeblikkelig. Rutinen kan benyttes til at undersøge, hvorvidt der skal udføres et synkront event. Se KL NEXT SYNC for at få denne rutine ind i dens rette sammenhæng.

Kapitel 22

Vi vil i dette kapitel lave et ur-program. Uret skal ikke være digitalt, men derimod med skive og visere. Programmet vil være interruptstyret (eller egentlig eventstyret) for at få det til at gå så præcist som muligt.

Vi vil nu gennemgå programmet. Bagest i kapitlet er den endelige udskrift af det.

Vi skal benytte et frame flyback interrupt, og dette event skal bruge en eventblok på 9 bytes. Disse lægges i starten af programmet. Desuden skal vi bruge 22 bytes til lyde – 2 blokke til SOUND QUEUE-rutinen og 1 blok til SOUND TONE ENVELOPE. Dernæst skal vi bruge 6 bytes, som fortæller noget om, hvor sekund-, minut- og timeviseren er placeret. Desuden skal vi bruge en byte, som holder styr på vort event. Vi bruger samme metode som i kapitel 18, hvor vi decrementerer en tæller (med startværdi 50) for at afgøre, hvornår viserne skal rykke sig. Der skal også benyttes en byte, der fortæller, om timeviseren skal flyttes. Sekundviseren skal naturligvis flyttes hvert sekund og minutviseren hvert minut, men timeviseren skal *ikke* kun flyttes hver time! Alle disse her nævnte reserverede bytes fylder de 39 første i programmet. Til sidst i programmet er placeret en enorm datablok på i alt 240 bytes, der indeholder 60 informationer à 4 bytes. Hver af de 4 bytes indeholder et 1. og 2. koordinat, der fortæller, hvordan viserne kan se ud. Derfor vil alle koordinaterne danne en cirkelskive, hvis der tegnes en streg gennem alle punkterne, som koordinaterne danner. Der er selvfølgelig 60 koordinatsæt, da viserne kan stå på 60 måder – der går jo 60 sekunder på et minut og 60 minutter på en time. Det første koordinatpar (placeret fra 40492) er den lodrette opadgående viser, der viser kl. 12.00. Når uret startes er klokken netop dette, så der fås

```
LD A,1
CALL SET SCR MODE
LD HL,40492
LD (sekundviser),HL
LD (minutviser),HL
LD (timeviser),HL
```

HL LOAD'es med 40492, da alle koordinaterne, som sagt ovenfor, starter på denne adresse. Bemærk også, at der arbejdes i MODE 1. Desuden har vi benyttet de såkaldte symbolske adresser, så vi bedre kan forstå programmet. Symbolske adresser angiver ikke selve adressen, men blot dens formål. På 2 bytes skulle gemmes tællere til hhv. vort event og til timeviser. Startværdierne sættes således:

```
LD HL,1586 (6 * 256 + 50)
LD (event- og timetæller),HL
```

Vi skal også sætte skærmkanten (BORDER) og baggrunden (pen 0) til sort, der er ink 0.

```
LD BC,0
PUSH BC
CALL SCR SET BORDER
POP BC
XOR A
CALL SCR SET INK
```

Vi skal i dette program bruge én af Amstrad's geniale muligheder, som åbenbares ved kald af rutinen SCR ACCESS. Hvis A indeholder 1, når rutinen kaldes, opnås den grafiske udskrivningstilstand, der kaldes XOR-MODE. Det betyder, at hvis vi plotter ovenpå et punkt sat med pen 2 med pen 3, vil den farve, som punktet opnår blive

```
    1 0  = 2
XOR 1 1  = 3
-------------
    0 1  = 1
```

tegnet med pen 1. Det vil også betyde, at hvis vi tegner oveni noget med samme pen, som det er tegnet med, vil vi opnå, at farven fra før de to plotninger genetableres. Hvis dette er baggrundsfarven med pen 0, vil to plotninger med samme farve i XOR-mode medføre, at der atter bliver pen 0's farve i punktet.

```
LD A,1
CALL SCR ACCESS
LD A,1
CALL GRA SET PEN
```

Samtidig sættes den grafiske pen til pen 1, i vores tilfælde gul. Nu skal vi have udskrevet tallene fra 1 til 12 i en cirkel. Til dette formål kan vi bruge koordinaterne i den lange datablok. Først skal origo dog sættes:

```
LD DE,316
LD HL,206
CALL GRA SET ORIGIN
```

Vi skal ikke bruge alle koordinatparrene i datablokken – kun 12 af dem, et til hvert tal. Med ialt 60 koordinatsæt skal vi benytte hvert femte for at få tallene spredt ligeligt rundt på cirklen. Vi lader IX pege på det koordinatpar, som angiver, hvor »1« skal stå. Det er starten af blokken + 4 * 5 = 40492 + 20 = 40512.

```
LD IX,40512
LD BC,2817 (11 * 256 + 1)
OM
PUSH BC
```

B LOAD'es med 11, antallet af tal, der skal udskrives (»12« udskrives senere), og C LOAD'es med 1. Dette er en tæller, som angiver, hvilket tal der næste gang skal udskrives.

Vi skal nu have flyttet koordinaterne over i DE og HL med førstnævnte indeholdende 1. koordinatet. Herefter ændres grafikpositionen.

```
LD E,(IX + 0)
LD D,(IX + 1)
LD L,(IX + 2)
LD H,(IX + 3)
CALL GRA MOVE ABSOLUTE
```

Vi henter nu BC ned (og gemmer den igen) for at finde ud af, hvad næste tal der skal udskrives er. Det ligger i C og skal flyttes til A.

```
POP BC
PUSH BC
LD A,C
```

Derefter følger en lille test.

```
CP 10
JR C, ET TAL
PUSH AF
LD A,49
CALL GRA WR CHAR
POP AF
SUB 10
ET TAL
ADD 48
CALL GRA WR CHAR
```

Der undersøges, om A er større end 9. Hvis dette ikke er tilfældet, hoppes der. Hvis det derimod er tocifret, udskrives først et ettal (tallet er jo enten 10 eller 11). Derefter fratrækkes 10, så vi tilbage har 0 eller 1. Til sidst tillægges 48 for at finde karakterkoden. Læg mærke til, at der udskrives på grafikpositionen.

Vi skal nu springe 4 koordinatpar over for at finde det næste tals position på skærmen. IX skal således tillægges 20.

```
LD BC,20
ADD IX,BC
```

```
POP BC
INC C
DJNZ OM
```

Til sidst finder vi ud af, om der er flere tal, og der hoppes tilbage, hvis dette er tilfældet. Nu resterer kun udskriften af »12«. Koordinatparret for dette ligger fra adresse 40492-40495.

```
LD DE,(40492)
LD HL,(40494)
CALL GRA MOVE ABSOLUTE
LD A,49
CALL GRA WR CHAR
LD A,50
CALL GRA WR CHAR
```

Vi vil også gerne have punkter alle de steder, hvor viserne kan komme, så vi i alt får en hullet cirkel. Der skal ialt sættes 60 prikker.

```
LD DE,320
LD HL,200
CALL GRA SET ORIGIN
LD IX,40492
LD B,60
NÆSTE
PUSH BC
LD E,(IX + 0)
LD D,(IX + 1)
LD L,(IX + 2)
LD H,(IX + 3)
CALL GRA PLOT ABSOLUTE
LD BC,4
ADD IX,BC
POP BC
DJNZ NÆSTE
```

Vi skal nu have lavet vores tre visere. Til sekundviseren bruger vi pen 3 (rød), til minutviseren pen 2 (den lyseblå) og pen 1 (gul) til

timeviseren. Minut- og sekundviseren er lige lange, mens timeviseren kun skal være halvt så lang. Til udskrivning af viserne bruger vi derfor dette:

```
LD BC,769 (3 * 256 + 1)
IGEN
PUSH BC
LD A,C
CALL GRA SET PEN
LD DE,0
PUSH DE
LD HL,0
CALL GRA MOVE ABSOLUTE
POP DE
LD HL,(40492)
LD A,1
CP C
LD A,L
CALL Z,FORKORT
CALL GRA LINE ABSOLUTE
POP BC
INC C
DJNZ IGEN
```

I starten LOAD'es B med antallet af viser og C med pennens farve. Så sættes pennen, og vi flytter grafikpositionen til (0,0). Derefter LOAD'es DE og HL med »kl. 12«-visernes koordinater. Imidlertid skal timeviseren jo halveres. Det er netop timeviseren, når C-registrets værdi er 1 (pennen). Hvis det ikke er timeviseren, tegnes viseren uden problemer, ellers kaldes rutinen FORKORT, der ser således ud:

```
BIT 7,H
JR Z,8
NEG
SRL A
NEG
JR 2
SRL A
```

```
LD L,A
RET
```

Først finder vi ud af, om HL er negativ. Dette afgøres af bit 15 i HL (det er bit 7 i H). Hvis denne bit er resat, kan vi halvere L, som er kopieret i A, med SRL A og hente A tilbage i L. Hvis HL derimod er negativ, skal vi tage vort tal opfattet to-komplementært, finde to-komplementet med NEG, halvere det, for igen at finde to-komplementet for at komme tilbage. Til sidst returneres til hovedprogrammet. Når viseren er tegnet i hovedprogrammet, incrementeres C for at pege på næste pen, og der hoppes tilbage, hvis der er flere visere.

Vi skal også give systemet besked om, at vi bruger en tone envelope.

```
LD HL,ent
LD A,1
CALL SOUND TONE ENVELOPE
```

Inden uret startes, skal du trykke på <ENTER>:

```
LD A,18
CALL KM TEST KEY
JR Z,- 7
```

Hvis den er nedtrykket, giver Amstrad besked om en eventrutine:

```
LD HL,eventblok
LD DE,eventrutine
LD BC,33024 (81h * 256)
CALL KL NEW FRAME FLY
```

C-registrets værdi er faktisk underordnet, da vi benytter en 2 bytes-adresse, se evt. vort kapitel om events. Fra nu af vil uret gå lige til det får en meddelelse om at stoppe. Hver gang der er gået 1/50 sekund kaldes eventrutinen. Hvis du ønsker at stoppe uret, må du trykke på <ESC>.

```
LD A,66
CALL KM TEST KEY
JR Z, - 7
LD HL,eventblok
CALL KL DEL FRAME FLY
RET
```

CALL-instruktionen sørger for, at vores event slettes fra listen over dem, der skal udføres ved hvert frame flyback interrupt.

Nu skal vi have lavet selve eventrutinen, der flytter viserne. Det er altså det egentlige program. Det forrige var blot initialiseringen, set up'et. Først i eventrutinen kommer dette:

```
LD HL,(eventtæller)
DEC (HL)
RET NZ
LD (HL),50
```

Rutinen kaldes jo hver 1/50 sekund, så derfor er vi – som nævnt i indledningen – nødt til at tælle en tæller ned 50 gange for at få flyttet viserne hvert sekund. Først skal sekundviserlyden udsendes.

```
LD HL,sekundlyd
CALL SOUND QUEUE
```

På adresse <sekundviser> står adressen på det koordinatpar i datablokken, som vores sekundviser for øjeblikket har. Denne skal slettes, og den nye tegnes. Den slettes ved blot at tegne en nøjagtig magen til oveni (vi har jo allerede fortalt, hvordan det gøres med den grafiske udskrivningstilstand 1).

```
LD IX,(sekundviser)
CALL SUB
CALL GRA LINE RELATIVE
```

Underrutinen SUB ser således ud:

```
PUSH IX
LE DE,0
LD HL,0
CALL GRA MOVE ABSOLUTE
POP IX
LD E,(IX + 0)
LD D,(IX + 1)
LD L,(IX + 2)
LD H,(IX + 3)
LD BC,4
ADD IX,BC
RET
```

Denne sørger blot for at få koordinaterne lagt i DE og HL samt at addere IX med 4, så vi får adressen på det næste koordinatpar. Når viseren er tegnet, hentes IX-registret over i HL med

```
PUSH IX
POP HL
```

Hvis L indeholder 28, vil den sidste adresse i datablokken være brugt, og det betyder, at der er gået et minut. Hvis det er tilfældet, skal minutviseren således flyttes.

```
LD A,L
CP 28
JR NZ,7
CALL MINUT
LD IX,40492
LD (sekundviser),IX
```

Hvis L ikke var 28, hoppes der 7 bytes frem, og dermed kaldes MINUT ikke og IX LOAD'es ikke med 40492. Hvis MINUT var kaldt, var datablokken jo slut, og adresse sekundviser måtte atter LOAD'es med blokkens startadresse.

Til sidst laves den nye sekundviser på denne måde:

```
LD A,3
CALL GRA SET PEN
```

```
CALL SUB
CALL GRA LINE RELATIVE
RET
```

Rutinen SUB er jo gennemgået på forrige side.

Vi vil ikke gennemgå rutinen MINUT, da denne fungerer på samme måde som med sekundviseren, selvfølgelig arbejdes blot med minutviseren. Desuden finder den ud af, om timeviseren skal rykkes. Det skal den hvert 12. minut! Da kaldes rutinen TIME. Du kan selvfølgelig se rutinen MINUT i det samlede program, der følger efter denne gennemgang.

Her følger rutinen TIME. Først skal den omtalte tæller stilles tilbage til 12 (minutter), og HL indeholder allerede adressen på tælleren fra rutinen MINUT.

```
LD (HL),12
LD A,1
CALL GRA SET PEN
LD IX,(timeviser)
CALL TVIS
```

Pennen sættes, og koordinaternes adresse findes, hvorefter rutinen TVIS kaldes. Den indeholder

```
CALL SUB
LD A,L
CALL FORKORT
PUSH HL
LD A,E
LD H,D
CALL FORKORT
POP DE
EX DE,HL
CALL GRA ABSOLUTE
RET
```

Rutinen TVIS kalder først SUB for at finde viserens koordinater. SUB-rutinen er forklaret tidligere. Så kalder vi FORKORT for at halvere viserens længde. Det gøres både med 1. og 2.

koordinatet. FORKORT kræver koordinatets MSB i H og LSB i A, så derfor benyttes

```
LD A,E
LD H,D
```

som det ses ovenfor. Rutinen FORKORT igen, og i HL har vi nu 1. koordinatet og på stakken 2. koordinatet. Med

```
POP DE
ED DE,HL
```

som det ses sørger vi for at 1. koordinatet havner i DE og 2. koordinatet i HL. Den timeviser, der i forvejen stod der, skal slettes ved at kalde GRA LINE ABSOLUTE. Dette skyldes atter vor specielle udskrivningstilstand. Dernæst skal den nye viser tegnes, og igen sker det på præcis samme måde som i MINUT og SEKUND, bortset fra viserens halvering. Det samlede program, som du skal indtaste fra adresse 40000 ser således ud:

Hex-kode	**Mnemonic**
0000000000	DEFB – EVENT
00000000	DEFB
01640201	DEFB – ENT –
0100002800	DEFB – SOUND SEK –
00070100	DEBF
0400010000	DEFB – SOUND MINUT –
00076400	DEFB
0000	DEFB – sekundviser –
0000	DEFB – minutviser –
0000	DEFB – timeviser –
00	DEFB – eventtæller –
00	DEFB – timetæller –
INIT – 40039	
3E01	LD A,1
CD0EBC	CALL SCR SET MODE
212C9E	LD HL,40492
225F9C	LD (40031),HL
22619C	LD (40033),HL

```
22639C          LD (40035),HL
213206          LD HL,1586
22659C          LD (40037),HL
010000          LD BC,0
C5              PUSH BC
CD38BC          CALL SCR SET BORDER
C1              POP BC
AF              XOR A
CD32BC          CALL SCR SET INK
3E01            LD A,1
CD59BC          CALL SCR ACCESS
3E01            LD A,1
CDDEBB          CALL GRA SET PEN
TAL – 40084
113C01          LD DE,316
21CE00          LD HL,206
CDC9BB          CALL GRA SET ORIGIN
DD21409E        LD IX,40512
01010B          LD BC,2817
OM
C5              PUSH BC
DD5E00          LD E,(IX + 0)
DD5601          LD D,(IX + 1)
DD6E02          LD L,(IX + 2)
DD6603          LD H,(IX + 3)
CDC0BB          CALL GRA MOVE ABSOLUTE
C1              POP BC
C5              PUSH BC
79              LD A,C
FE0A            CP 10
3809            JR C,ET TAL
F5              PUSH AF
3E31            LD A,49
CDFCBB          CALL GRA WR CHAR
F1              POP AF
D60A            SUB 10
ET TAL
C630            ADD 48
CDFCBB          CALL GRA WR CHAR
```

```
011400      LD BC,20
DD09        ADD IX,BC
C1          POP BC
0C          INC C
10D2        DJNZ OM
ED5B2C9E    LD DE,(40492)
2A2E9E      LD HL,(40494)
CDC0BB      CALL GRA MOVE ABSOLUTE
3E31        LD A,49
CDFCBB      CALL GRA WR CHAR
3E32        LD A,50
CDFCBB      CALL GRA WR CHAR
PLOTS – 40166
114001      LD DE,320
21C800      LD HL,200
CDC9BB      CALL GRA SET ORIGIN
DD212C9E    LD IX,40492
063C        LD B,60
NÆSTE
C5          PUSH BC
DD5E00      LD E,(IX + 0)
DD5601      LD D,(IX + 1)
DD6E02      LD L,(IX + 2)
DD6603      LD H,(IX + 3)
CDEABB      CALL GRA PLOT ABSOLUTE
010400      LD BC,4
DD09        ADD IX,BC
C1          POP BC
10E8        DJNZ NÆSTE
VISERE – 40205
010103      LD BC,769
IGEN
C5          PUSH BC
79          LD A,C
CDDEBB      CALL GRA SET PEN
110000      LD DE,0
D5          PUSH DE
210000      LD HL,0
CDC0BB      CALL GRA MOVE ABSOLUTE
```

```
D1              POP DE
2A2E9E          LD HL,(40494)
3E01            LD A,1
B9              CP C
7D              LD A,L
CC1C9E          CALL Z,FORKORT
CDF6BB          CALL GRA LINE ABSOLUTE
C1              POP BC
0C              INC C
10DF            DJNZ IGEN
ENT – 40241
21499C          LD HL,40009
3E01            LD A,1
CDBFBC          CALL SOUND TONE ENVELOPE
START
3E12            LD A,18
CD1EBB          CALL KM TEST KEY
28F9            JR Z, – 7
EVENT
21409C          LD HL,40000
115A9D          LD DE,40282
010081          LD BC,33024
CDD7BC          CALL KL NEW FRAME FLY
SLUT
3E06            LD A,6
CD1EBB          CALL KM TEST KEY
28F9            JR Z, – 7
21409C          LD HL,40000
CDDDBC          CALL KL DEL FRAME FLY
C9              RET
EVENTRUT – 40282
21659C          LD HL,40037
35              DEC (HL)
C0              RET NZ
3632            LD (HL),50
SEK
214D9C          LD HL,40013
CDAABC          CALL SOUND QUEUE
DD2A5F9C        LD IX,(40031)
```

CDFD9D	CALL SUB
CDF6BB	CALL GRA LINE ABSOLUTE
DDE5	PUSH IX
E1	POP HL
7D	LD A,L
FE1C	CP 28
2007	JR NZ,7
CD909D	CALL MINUT
DD212C9E	LD IX,40492
DD225F9C	LD (40031),IX
3E03	LD A,3
CDDEBB	CALL GRA SET PEN
CDFD9D	CALL SUB
CDF6BB	CALL GRA LINE ABSOLUTE
C9	RET

MINUT – 40336

21569C	LD HL,40022
CDAABC	CALL SOUND QUEUE
3E02	LD A,2
CDDEBB	CALL GRA SET PEN
DD2A619C	LD IX,(40033)
CDFD9D	CALL SUB
CDF6BB	CALL GRA LINE ABSOLUTE
DDE5	PUSH IX
DDE5	PUSH IX
21669C	LD HL,40038
35	DEC (HL)
CCCC9D	CALL Z,TIME
DDE1	POP IX
E1	POP HL
7D	LD A,L
FE1C	CP 28
2004	JR NZ,4
DD212C9E	LD IX,40492
DD22619C	LD (40033),IX
3E02	LD A,2
CDDEBB	CALL GRA SET PEN
CDFD9D	CALL SUB
CDF6BB	CALL GRA LINE ABSOLUTE

```
C9              RET
TIME – 40396
360C            LD (HL),12
3E01            LD A,1
CDDEBB          CALL GRA SET PEN
DD2A639C        LD IX,(40035)
CDEA9D          CALL TVIS
DDE5            PUSH IX
E1              POP HL
7D              LD A,L
FE1C            CP 28
2004            JR NZ,4
DD212C9E        LD IX,40492
DD22639C        LD (40035),IX
TVIS – 40426
CDFD9D          CALL SUB
7D              LD A,L
CD1C9E          CALL FORKORT
E5              PUSH HL
7B              LD A,E
62              LD H,D
CD1C9E          CALL FORKORT
D1              POP DE
EB              EX DE,HL
CDF6BB          CALL GRA LINE ABSOLUTE
C9              RET
SUB – 40445
DDE5            PUSH IX
110000          LD DE,0
210000          LD HL,0
CDC0BB          CALL GRA MOVE ABSOLUTE
DDE1            POP IX
DD5E00          LD E,(IX + 0)
DD5601          LD D,(IX + 1)
DD6E02          LD L,(IX + 2)
DD6603          LD H,(IX + 3)
010400          LD BC,4
DD09            ADD IX,BC
C9              RET
```

FORKORT – 40476

CB7C	BIT 7,H
2808	JR Z,8
ED44	NEG
CB3F	SRL A
ED44	NEG
1802	JR 2
CB3F	SRL A
6F	LD L,A
C9	RET

DATA – 40492

0000B90013	DEFB
00B8002600	DEFB
B5003900B0	DEFB
004B00A900	DEFB
5D00A0006D	DEFB
0096007C00	DEFB
890089007C	DEFB
0096006D00	DEFB
A0005D00A9	DEFB
004B00B000	DEFB
3900B50026	DEFB
00B8001300	DEFB
B9000000B8	DEFB
00EDFFB500	DEFB
DAFFB000C7	DEFB
FFA900B5FF	DEFB
A000A4FF96	DEFB
0093FF8900	DEFB
84FF7C0077	DEFB
FF6D006AFF	DEFB
5D0060FF4B	DEFB
0057FF3900	DEFB
50FF26004B	DEFB
FF130048FF	DEFB
000047FFED	DEFB
FF48FFDAFF	DEFB
4BFFC7FF50	DEFB
FFB5FF57FF	DEFB

```
A4FF60FF93      DEFB
FF6AFF84FF      DEFB
77FF77FF84      DEFB
FF6AFF93FF      DEFB
60FFA4FF57      DEFB
FFB5FF50FF      DEFB
C7FF4BFFDA      DEFB
FF48FFEDFF      DEFB
47FF000048      DEFB
FF13004BFF      DEFB
260050FF39      DEFB
0057FF4B00      DEFB
60FF5D006A      DEFB
FF6D0077FF      DEFB
7C0084FF89      DEFB
0093FF9600      DEFB
A3FFA000B5      DEFB
FFA900C7FF      DEFB
B000DAFFB5      DEFB
00EDFFB800      DEFB
```

Når du kører programmet med CALL 40039, vil du se urskiven og tallene, men du vil ikke se nogle visere, kun en lille lodret, gul streg. Tryk på <ENTER>, og uret går i gang. Nu vil du se sekundviseren og en lodret streg i to forskellige farver. Vent til et minut er gået, og du vil se minutviseren bevæge sig.

Grunden, til at viserne i starten ser så mærkelige ud, er, at vi bruger den helt specielle udskrivningstilstand XOR-MODE.

Programmet er konstrueret, så det er nogenlunde let at forstå, og den mest pladsbesparende metode er ikke brugt. Du kan selvfølgelig selv prøve at forbedre det hvis du får lyst.

Kapitel 23

En af Amstrad's rigtig fine egenskaber er, at man kan definere sine egne BASIC-kommandoer. Vi vil komme med flere eksempler i dette kapitel. Det første er en kommando EXPLODE, som vil give en lyd. Prøv at se på dette program:

Hex-kode	**Mnemonic**	**Adresse**
014E9C	LD BC,40014	40000
214A9C	LD HL,40010	40003
CDD1BC	CALL KL LOG EXT	40006
C9	RET	40009
00000000	DEFB – SYSTEM	40010
539C	DEFB –TABEL –	40014
C35B9C	JP EXPLODE	40016
4558504C4F	DEFB E,X,P,L,O	40019
44C5	DEFB D,(E + 128)	40024
00	DEFB 0	40026
EXPLODE		
216A9C	LD HL,40042	40027
3E01	LD A,1	40030
CDBCBC	CALL SOUND AMPL ENVELOPE	40032
216E9C	LD HL,40046	40035
CDAABC	CALL SOUND QUEUE	40038
C9	RET	40041
010FFF0A	DEFB – ENVELOPE –	40042
0701000000	DEFB – SOUND –	40046
110F9600	DEFB – SOUND –	40051

BC bliver i starten LOAD'et med 40014, og fra den adresse står to bytes, der angiver startadressen på en tabel indeholdende alle de nye kommandoer – der kan nemlig godt defineres mere end én. Efter de to bytes står der en JP-instruktion, som fortæller Am-

strad, at hvis vores kommando er fundet i et program, skal den hoppe til den adresse, som JP-instruktionen angiver. Fra adresse 40019 står vores kommando EXPLODE lagret i 7 bytes. Læg mærke til, at det er de store bogstavers kode som er lagret, og bemærk endvidere, at det sidste bogstav i kommandoen (det er E) er lagret som C5h. Efter undersøgelser vil du se, at det er koden for »E«, men bit 7 er sat. Denne bit sættes for at angive, at kommandoen ikke er længere. På adresse 40026 står der 0, hvilket angiver, at der ikke er flere kommandoer i tabellen. I starten af programmet bliver HL LOAD'et med 40010. Det skyldes, at systemet skal bruge 4 bytes til forskellige gøremål, når ekstra kommandoer benyttes. HL peger på den adresse, hvorfra de 4 bytes er reserveret af os. CALL-instruktionen giver Amstrad besked om den ekstra kommando, så den kender den. Selve programmet, som giver lyden, burde ikke volde dig vanskeligheder forståelsesmæssigt.

Efter indtastning skal du i BASIC skrive CALL 40000. For at afprøve kommandoen skal du først nedtrykke <SHIFT> og tasten til højre for P samtidigt. Dette tegn er en lodret streg. *Alle* selvdefinerede kommandoer skal begynde med dette tegn. Indtast lige efter tegnet ordet EXPLODE og tryk <ENTER>. Lyden skulle komme ud fra den indbyggede højttaler.

Det er også muligt at have parametre efter en kommando. ROM er så venlig, at A-registret vil indeholde antallet af parametre, mens IX vil indeholde startadressen på en tabel, hvor alle parametrene står. Disse parametre er altid enkeltvis lagret i to bytes, altså et tal mellem 0 og 65535. Hvis der kun er én parameter, er DE LOAD'et med denne. Hvis der før parameteren er placeret tegnet til højre for P-tasten (uden SHIFT), er DE adressen, hvor parameteren står – stadig i to bytes.

Hvis der er flere, vil IX som sagt angive en adresse. Fra denne adresse er parametrene lagt på en bagvendt måde. Den sidste parameter er lagret først, og som sædvanlig står LSB af parameteren først og MSB sidst. IX vil altid pege på LSB af den sidste parameter.

Hvis der er parametre efter kommando, skal de som normalt adskilles af kommategn, men der skal også være sat et kommategn mellem kommandoen og den første parameter.

Lad os prøve at lave kommandoen SCREEN, som skal finde ud

af hvilket tegn, der står på printpositionen. Denne kommando mangler Amstrad. Prøv dette program:

Hex-kode	**Mnemonic**
014E9C	LD BC,40014
214A9C	LD HL,40010
CDD1BC	CALL KL LOG EXT
C9	RET
00000000	DEFB – SYSTEM –
539C	DEFB – KOMMANDOTABEL –
C35A9C	JP SCREEN
53435245	DEFB – S,C,R,E –
45CE	DEFB – E, (N + 128) –
00	DEFB 0
SCREEN	
D5	PUSH DE
CD60BB	CALL TXT READ CHAR
D1	POP DE
12	LD (DE),A
C9	RET

Det første i programmet er det samme som i vort EXPLODE-program. Selve SCREEN-kommandoen starter med PUSH DE. Det er meningen, at der efter SCREEN-kommandoen skal være én heltalsvariabel som parameter. Denne parameter skal starte med tegnet **@**. DE vil altså komme til at indeholde adressen for stedet, hvor vor heltalsvariabel er lagret. Derefter kaldes TXT READ CHAR, og herefter vil der i A stå indholdet og printpositionen. A lægges nu i vores heltalsvariabel.

Kør programmet med

```
CLS : CALL 40000
```

Der vil nu stå et R i øverste venstre hjørne. Skriv så

```
LOCATE 1,1 : A % = 0 : | SCREEN, @A % <ENTER>
PRINT CHR $ (A %) <ENTER>
```

Du vil få et R udskrevet – som ventet. Husk at A %-variablen skal være defineret inden SCREEN-kommandoen eksekveres, ellers ved Amstrad ikke, hvor den har gjort plads til A %, og dermed ikke hvor den skal lægge tallet, vi får fra SCREEN.

Lad os nu prøve at lave et program, hvor flere parametre bruges. Vi kan f.eks. lave kommandoen HEX, som skal udskrive RAM i hexadecimal notation. Første parameter angiver, hvorfra der skal udskrives, og anden parameter hvor mange bytes der er tale om.

Hex-kode	**Mnemonic**
014E9C	LD BC,40014
214A9C	LD HL,40010
CDD1BC	CALL KL LOG EXT
C9	RET
00000000	DEFB – SYSTEM –
539C	DEFB – KOMMANDOTABEL –
C3579C	JP HEX
4845D8	DEFB – H,E,(X + 128) –
00	DEFB 0
HEX	
FE02	CP 2
C0	RET NZ
DD6603	LD H,(IX + 3)
DD6E02	LD L,(IX + 2)
DD4600	LD B,(IX + 0)
E1	
7E	LD A,(HL)
0E02	LD C,2
57	LD D,A
1F	RRA
1F	RRA
1F	RRA
1F	RRA
E2	
E60F	AND 15
FE0A	CP 10
3802	JR C,2
C607	ADD 7

```
C630        ADD 48
CD5ABB      CALL TXT OUTPUT
7A          LD A,D
0D          DEC C
20EF        JR NZ,E2
3E20        LD A,32
CD5ABB      CALL TXT OUTPUT
CD5ABB      CALL TXT OUTPUT
23          INC HL
10DC        DJNZ E1
C9          RET
```

Det første af programmet indtil rutinen HEX kender du. Med CP 2 og RET NZ laver vi en slags syntakscheck. Husk, at A indeholder antallet af parametre. Hvis der ikke er angivet to, vil programmet returnere uden at der er sket noget. Vi kunne selvfølgelig have fået den til at skrive SYNTAX ERROR.

IX vil pege på en tabel, hvor de to parametre står. Den første er lagret sidst, hvorfor H og L bliver LOAD'et med hhv. (IX + 3) og (IX + 2). B bliver kun LOAD'et med LSB af den anden parameter. Dette betyder, at der højst vil blive udskrevet 256. Hvis 2. parameter er 280, vil der udskrives 24 – LSB af 280.

Selve udskrivningen af hex-tallene skulle du kunne forstå. Det er nemlig et program, som vi har gennemgået i kapitel 11.

Kør programmet med CALL 40000, og prøv så at skrive

```
| HEX,40000,72 (ENTER)
```

Du vil hermed få hele programmet udskrevet!

Vi har altså mulighed for at lægge hele programmer ned i en enkelt kommando. Lad os nu prøve et program med flere nye kommandoer. Vi ønsker et udvalg af SCROLL-kommandoer – du kan naturligvis selv udvide eller ændre dette, når du har gennemarbejdet vores forslag. For det første vil vi lave kommandoen VENSTRE, som kører de rækker, som parametrene angiver til venstre. Det som forsvinder i venstre side, skal dukke op til højre.

En tilsvarende HOJRE-kommando skal laves. Vi bruger et O, fordi vi ingen Ø har på Amstrad – du kan selvfølgelig selv define-

re ét. Vi vil også have kommandoen OP, som ruller hele skærmen op. Hvis der er angivet 4 parametre, er det kun et vindue, der skal rulles op. Parametrene skal angives som ved WINDOW-kommandoen. Vi skal desuden bruge en NED-kommando. Funktionen af denne behøver vi næppe gøre rede for her.

Det første, der skal ske i programmet, er at give computeren besked om de 4 nye kommandoer. Du kan se, hvorledes det gøres i den samlede opskrivning af programmet, der følger. Efter dette skal laves kommandoerne OP og NED. Vi har jo to rutiner i LROM, som kan ordne det for os – en ting, som Amstrad's BASIC ikke udnytter! Den ene hedder SCR HW ROLL, og den ruller hele skærmen, mens SCR SW ROLL kun ruller et vindue. Register B skal LOAD'es med 0, hvis der skal rulles ned, og med ikke-nul, hvis der skal rulles op. Vi tager NED og OP på en gang:

```
NED
LD B,0
JR 2
OP
LD B,255
CP 0
JP Z, SCR HW ROLL
```

Den eneste forskel på OP og NED er B-registrets indhold. 0 er LOAD'et for NED, og 255 for OP. (Overvej i øvrigt at forbedre denne rutine). Herefter finder vi ud af, om register A er nul, hvilket betyder, at der ingen parametre er. Hvis der er dette, hoppes ikke, men fortsættes blot videre i programmet. Hvis der ingen parametre er, hoppes der til rutinen SCR HW ROLL, der scroller. Måske har du i kapitel 21 set, at A skal LOAD'es med en farvekode. Vi skal i dette program bruge kode 0, fordi den nye linje skal være i baggrundsfarven, men A er jo allerede nul, så vi sparer en LD A,0 eller XOR A. Du undrer dig vel over, at der står JP i stedet for CALL. Det er fordi, at vi så slipper for en RET i vort program. Når LROM støder på en RET, vil der fortsættes i BASIC og ikke returneres til vort program.

Hvis der var parametre, vil der fortsættes herunder:

```
CP 4
RET NZ
```

Hvis der ikke er 4 parametre, skal der intet ske, og vi returnerer. Vi skal jo bruge 4 for at definere et vindue. Igen skal vi kun bruge LSB af parametrene, da den højeste parameter vil være 80 (der er jo højst 80 kolonner – i mode 2).

```
LD H,(IX + 6)
LD D,(IX + 4)
LD L,(IX + 2)
LD E,(IX + 0)
```

Se efter i kapitel 21, at parametrene er havnet i de rigtige registre. Husk, at i tabellen står den sidste parameter først. Hvis du læser vor gennemgang af rutinen grundigt igennem, vil du opdage, at denne rutine betragter øverste, venstre hjørne som 0,0, mens vi bruger koordinaterne 1,1. Det må vi kompensere for:

```
DEC H
DEC D
DEC L
DEC E
```

Den nye linje, der skal fremkomme, skal igen være i baggrundsfarven.

```
XOR A
CALL SCR SW ROLL
RET
```

Så er rutinerne til OP og NED i orden. Det var temmelig enkelt. Det er straks værre med VENSTRE og HOJRE, fordi skærmbilledet har den mærkelige opbygning, hvilket vi allerede har forklaret i kapitel 15. For at rulle en hel række til siden, er vi desværre nødt til at tage os af hver af de 8 tynde, vandrette linjer, som en række består af. Vi er altså nødt til at rulle 80 bytes 8 gange (en linje er jo netop 80 bytes lang). Et andet krav, vi vil stille, er, at øverste venstre hjørne svarer til adresse 49152. Dette kan opfyldes f.eks.

ved en MODE-kommando. Når en parameter er angivet, er vi – ifølge vor tidligere tegning af skærmopbygningen – nødt til at multiplicere den med 80 for at kende adressen på den enkelte rækkes øverste venstre hjørne. Derfor må vi først have lavet en rutine, som udfører dette. Parameteren bliver lagt i L, og sættes til nul.

HL skal således multipliceres med 80, men inden da skal L decrementeres. Dette sker, fordi vi angiver den øverste linje med parameteren 1. Denne har jo imidlertid en startadresse på 49152 + *0* * 80! Ialt får vi

```
MUL 80
LD H,0
LD L,(IX + 0)
DEC L
INC IX
INC IX
ADD HL,HL
ADD HL,HL
ADD HL,HL
ADD HL,HL
PUSH HL
POP DE
ADD HL,HL
ADD HL,HL
ADD HL,DE
RET
```

IX incrementeres to gange for at pege på LSB af den næste parameter, såfremt en sådan findes. Med alle ADD-instruktionerne multipliceres HL med 80, hvilket du vil kunne overbevise dig om ved en nøjere gennemgang.

Nu kommer VENSTRE-rutinen, hvor antallet af parametre startes med at blive gemt på stakken.

```
VENSTRE
PUSH AF
CALL MUL 80
```

```
LD DE,49152
ADD HL,DE
```

MUL 80 kaldes for at hente parameteren og multiplicere den med 80. DE LOAD'es med skærmens øverste venstre hjørne, og HL indeholder herefter adressen på rækkens øverste venstre hjørne. En række bestod af 8 linjer.

```
LD B,8
```

Herefter gemmes B (og C), og HL kopieres i DE:

```
IGEN
PUSH BC
PUSH HL
POP DE
```

Nu incrementerer vi HL.

```
INC HL
```

Nu peger DE på linjens første byte, mens HL peger på den anden. Nu kan vi flytte hele linjen én byte til venstre, men inden LOAD'es BC med linjens længde minus 1, og A med linjens første byte.

```
LD BC,79
LD A,(DE)
LDIR
LD (DE),A
```

Efter blokinstruktionen er hele linjen rykket, og DE peger på linjens sidste byte. DE blev jo tillagt 79 gennem LDIR-instruktionen. A indeholdt linjens yderste venstre byte, som med LD (DE),A flyttes til linjens højre side.

Nu skal næste linje i rækken flyttes. Næste linjes første adresse er adressen på den nuværende tillagt 2048. Umiddelbart skulle man tro, at HL skulle tillægges 2048, men den er jo blevet for-

øget undervejs. Først blev den incrementeret, og LDIR forøgede den med 79, så i alt fald skal HL nu tillægges 2048 – 80 = 1968.

```
LD BC,1968
ADD HL,BC
```

Til sidst henter vi BC ned igen for at finde ud af, om der er flere linjer i rækken.

```
POP BC
DJNZ IGEN
```

Så resterer kun at undersøge, om der er flere parametre i kommandoen. Hvis dette ikke er tilfældet, skal der returneres.

```
POP AF
DEC A
JR NZ, VENSTRE
RET
```

HØJRE-rutinen fungerer helt på samme måde, men vi bruger LDDR-instruktionen i stedet for LDIR.

Her er det færdige program, som du skal indtaste.

Hex-kode	**Mnemonic**
014E9C	LD BC,40014
214A9C	LD HL,40010
CDD1BC	CALL KL LOG EXT
C9	RET
00000000	DEFB – SYSTEM –
5C9C	DEFB – KOMMANDOTABEL –
C36E9C	JP NED
C3729C	JP OP
C3A59C	JP VENSTRE
C3C69C	JP HØJRE
4E45C4	DEFB – N,E,(D + 128) –
4FD0	DEFB – 0,(P + 128) –
56454E53	DEFB – V,E,N,(S + 128) –

```
5452C5          DEFB – T,R,(E + 128) –
484F4A52C5      DEFB – H,O,J,R,(E + 128) –
00              DEFB 0
NED
0600            LD B,0
1802            JR 2
OP
06FF            LD B,255
FE00            CP 0
CA4DBC          JP Z,SCR HW ROLL
FE04            CP 4
C0              RET NZ
DD6606          LD H,(IX + 6)
DD5604          LD D,(IX + 4)
DD6E02          LD L,(IX + 2)
DD5E00          LD E,(IX + 0)
25              DEC H
15              DEC D
2D              DEC L
1D              DEC E
AF              XOR A
CD50BC          CALL SCR SW ROLL
C9              RET
MUL 80
2600            LD H,0
DD6E00          LD L,(IX + 0)
2D              DEC L
DD23            INC IX
DD23            INC IX
29              ADD HL,HL
29              ADD HL,HL
29              ADD HL,HL
29              ADD HL,HL
E5              PUSH HL
D1              POP DE
29              ADD HL,HL
29              ADD HL,HL
19              ADD HL,DE
C9              RET
```

```
VENSTRE
F5          PUSH AF
CD919C      CALL MUL 80
1100C0      LD DE,49152
19          ADD HL,DE
0608        LD B,8
IGEN
C5          PUSH BC
E5          PUSH HL
D1          POP DE
23          INC HL
014F00      LD BC,79
1A          LD A,(DE)
EDB0        LDIR
12          LD (DE),A
01B007      LD BC,1968
09          ADD HL,BC
C1          POP BC
10EE        DJNZ IGEN
F1          POP AF
3D          DEC A
20E0        JR NZ, VENSTRE
C9          RET
HØJRE
F5          PUSH AF
CD919C      CALL MUL 80
114FC0      LD DE,49231
19          ADD HL,DE
0608        LD B,8
OM
C5          PUSH BC
E5          PUSH HL
D1          POP DE
2B          DEC HL
014F00      LD BC,79
1A          LD A,(DE)
EDB8        LDDR
12          LD (DE),A
015008      LD BC,2128
```

```
09          ADD HL,BC
C1          POP BC
10EE        DJNZ OM
F1          POP AF
3D          DEC A
20E0        JR NZ, HØJRE
C9          RET
```

Bemærk, at i rutinen HØJRE er DE i starten LOAD'et med den sidste byte i den allerførste linje. Bemærk også, at BC LOAD'es med 2128 = 2048 + 80 i stedet for 1968 i VENSTRE. Det skyldes selvfølgelig, at LDDR decrementerer HL, hvor LDIR incrementerer den.

Som allerede nævnt i starten af vor forklaring af programmet, kan du selv ændre det. Måske ønsker du ikke, at den karakter, der scrolles væk i VENSTRE (og HØJRE) skal dukke op i den modsatte side – måske vil du have, at den øverste linje ved OP skal dukke op for neden etc. Der er muligheder nok. Du kan også prøve at lægge helt andre features ind, så Amstrads BASIC bliver udbygget.

Vi kan afprøve vores nye kommandoer med dette BASIC-program:

```
 10  CALL 40000
 20  MODE 1
 30  FOR N = 1 TO 1000
 40  PEN (N) MOD 3 + 1 : PRINT " * ";
 50  NEXT
 60  WHILE N < 12
 70  |VENSTRE,1 : |VENSTRE,1
 80  |NED,1,1,1,25
 90  |HOJRE,25 : |HOJRE,25
100  |OP,40,40,1,25
110  |NED,2,39,2,13
120  |OP,2,39,13,24 : N = N + 1
130  WEND
140  LOCATE 14,2 : PRINT »MASKINKODE«
150  LOCATE 18,4 : PRINT »MED«
```

```
160  LOCATE 12,24 : PRINT »AMSTRAD CPC 464«
170  N = 0
180  GOTO 60
```

Kapitel 24

Det sidste program, vi vil bringe, er et såkaldt »singlestep«-program. Det betyder, at det tillader dig at køre et maskinkode-program instruktion for instruktion. Hvad nytte kan man drage af det? Hvis man har lavet et program, som ikke virker efter hensigten, kan det være svært at finde fejlen. Det er på dette sted, at vores program kommer ind i billedet. Hvis du »singlestepper« dig igennem det fejlagtige program, vil du kunne se, hvor det går galt – hvor Amstrad handler anderledes end du havde ventet. Efter udførelsen af hver instruktion vil indholdet af alle registrene blive skrevet ud både i decimal, hexadecimal og binær notation. Desuden vil du også kunne se hvilke værdier, der ligger på stakken. Det er dog ikke alle registrene, vi har valgt at få udskrevet. De alternative registre udskrives ikke, fordi de er temmelig besværlige i brug, hvilket du vel erfarede i kapitel 19. Desuden udskrives I- og R-registret ikke.

Som sagt kan vores program fortælle dig, hvor dit program gør noget andet, end du havde troet. Hvordan vores program virker, vil vi ikke forklare så grundigt, som vi har gjort det med de øvrige programmer. Programmet fylder nemlig i alt ca. 1,2 Kbytes, og en grundig forklaring ville være så drøj, at bogen skulle udkomme i flere bind! Vi vil her forklare, hvordan programmer set i grove træk virker uden at komme ind på de enkelte detaljer. Til gengæld forventer vi, at du selv går programmet igennem. Det vil du sikkert kunne lære meget af, da vi har bestræbt os på at bruge et så bredt udvalg af instruktioner som muligt.

Hvordan virker så programmet? Det første punkt er at finde ud af, hvilken instruktion der skal eksekveres. Til dette formål har vi udviklet rutinen NXT-INS beliggende i 41650-41783, begge adresser incl. Til denne rutine er knyttet den 128 bytes lange datablok, der ligger fra adresse 41984 til 42111. Disse data indeholder information om, hvor lange alle mulige instruktioner er, med undtagelse af ED-instruktionerne. Disse er nemlig altid 2

bytes lange med undtagelse af 8 instruktioner. Det drejer sig om ED43nn, ED4Bnn, ED53nn, ED5Bnn, ED63nn, ED6Bnn, ED73nn og ED7Bnn. Hvis vi nu ser på anden byte i disse instruktioner, kan vi iagttage, at de har en ting tilfælles. Byten er binært af denne form:

0 1 * * * 0 1 1

Det betyder, at ved at slette bit 5, 4 og 3 med en AND 11000111 (AND 199) vil vi have tallet 67 (64 + 2 + 1) tilbage. På den måde kan vi kontrollere, om der er tale om en 4 bytes-ED-instruktion.

De øvrige instruktioners længde står som sagt lagret i de 128 bytes i datablokken. Da en normal instruktion er mellem 1 og 3 bytes lang, kan vi nøjes med at bruge 2 bit pr. instruktion. Når en instruktions første hex-kode – også kalde OP-koden – kendes, slår vi simpelt hen op i tabellen og finder de pågældende 2 bits. Disse to bit isoleres ved AND-instruktion hvorefter hele instruktionen udskrives. På samme måde virker det med DD- og FD-instruktioner (operationer med IX- og IY-registrene). De er også mellem 1 og 3 bytes lange, hvis der ses bort fra DD og FD. I en halv byte (4 bit) er altså lagret en instruktions længde, hvis den er normal eller hvis det er en IX/IY-instruktion. Hvad så med CB-instruktionerne? De er altid 2 bytes lange, dog 4, hvis det er IX/IY-instruktioner. I den 203. halvbyte (CBh = 203) i tabellen står der på den måde følgende 4 bit: 1110 – splittet op giver det 3 og 2. FD og DD er som nævnt aldrig indkalkuleret i længden af vores tabel. Vores rutine ordner dette ved at udskrive DD/FD for sig selv. Det kan du selv studere grundigere.

Hvis du er opmærksom, vil du se, at alle bytes i en instruktion bliver lagt i nogle adresser. IX indeholder 41298, når NXT-INS-rutinen bruges. Der er således reserveret 4 bytes fra adresse 41298 til 41301 til den instruktion, som vi står for at eksekvere. Inden dette sker, bliver alle registrene LOAD'et med deres værdi efter den sidst udførte instruktion. Efter instruktionens udførelse bliver alle registrenes værdier gemt på adresserne 42112-42127. Læg i øvrigt mærke til, at vi ikke bruger den normale stak. Vi har simpelt hen oprettet vores egen, hvor SP ved programmets start er 41000. På adresserne 42124-42125 gemmes vores nye stak, mens den rigtige gemmes i 42126-42127.

Inden den mulige eksekvering af vor instruktion undersøger vi med rutinen fra adresse 41000, om instruktionen er af en art, som forårsager, at der hoppes i programmet. Disse instruktioner er JR, DJNZ, JP, CALL og RET. Når der skal udføres en JR, en DJNZ eller en JP, så bliver instruktionen ikke eksekveret, men der sørges for, at der »single-step'pes« næste gang fra det sted, som der egentlig skal hoppes til. Hvis der er tale om en DJNZ-instruktion sørger vi selvfølgelig for at decrementere B-registret. Hvis det er en CALL, »single-step'per« vi også næste gang fra den adresse, som CALL angiver, men vi husker samtidig at gemme returadressen på vores nyoprettede stak.

Med en enkelt undtagelse lader vi dog CALL-instruktionen udføre. Det er, når du bruger en rutine i jump-blokkene. Disse starter allerede i adresse B900h, så vi sammenligner MSB af CALL-instruktionen med B9h (185). Hvis det er et ROM-kald, udføres det uden forholdsregler. Ved udarbejdelsen blev vi enige om, at det ikke ville have nogen speciel interesse at se, hvad ROM indeholder. Hvis det f.eks. drejer sig om rutinen GRA LINE ABSOLUTE, vil det faktisk være en ganske stor samling instruktioner, som skal gennemkøres. Hvis det er et ROM-kald, vil vor stak således heller ikke blive berørt.

Når der mødes en RET, henter vi blot returadressen ned og »single-step'per« fra denne adresse næste gang. Hvis SP er 41000, når en RET skal eksekveres, returneres bare til BASIC. Hvis programmet ikke fandt nogle instruktioner, som krævede speciel behandling, eksekveredes rutinens næste instruktion af rutinen EXECUTE.

Derfor skal du sørge for, at der ikke bliver udført instruktioner, som får maskinen til at crashe, som f.eks. EXX. Når instruktionen er udført, skal vi have udskrevet alle registrenes indhold. Dette sker fra adresse 41418, hvor vi først får udskrevet alle 8 bits-registrene (undtagen F) i decimal, hexadecimal og binær notation. Fra adresse 41456 udskrives 16 bits-registrene, men ikke i binær notation. Hvis man ønsker at se dem i denne notation, kan man jo blot se på de to pågældende 8 bits-registre, som 16 bits-registret er sammensat af. Det betyder også, at IX, IY og SP ikke kan ses binært, men det er normalt heller ikke aktuelt, da der f.eks. ikke findes nogle bit-instruktioner (RES, SET, BIT m.fl.) til disse deciderede 16 bits-registre. Fra adresse 41508 udskrives flag-regi-

strets indhold, selvfølgelig kun binært. Fra adresse 41536 udskrives indholdet af noget af stakken. Med dette menes de 7 underste værdier. Grunden, til at der ikke udskrives flere, har med skærmens opbygning at gøre. Vi har bestræbt os på ved udarbejdelsen at gøre den så overskuelig som mulig. Det kan ordnes, således at flere værdier kan udskrives, men der er normalt begrænsninger for, hvor meget stakken benyttes. 7 værdier bør under almindelige omstændigheder være nok! Hvis SP peger på 40996, vil du kun få 2 værdier udskrevet, da bunden jo fra starten er blevet sat til 41000. Over denne adresse ligger faktisk hele vores program. Når registrene er udskrevet, er vi klar til at udføre næste instruktion.

I øvrigt bliver skærmen lige klargjort, inden vi begynder at »single-step'pe«. Registrenes navne, navnene på de enkelte flag o.lign. indgår i klargørelsen. Det sker på adresserne 41322-41417. Dette har en datablok tilknyttet, som ligger på 41888-41981. Denne blok beskriver således, hvad der skal stå på skærmen.

Programmet følger nu i sin helhed, og det er bare om at komme i gang med indtastningen! Eksempler på brugen af det følger efter programmet. Sørg iøvrigt for, at hex-loaderen starter fra adr. 41000.

SINGLESTEP – 41000

Hex-kode	**Mnemonic**	**Kommentarer**
DD2152A1	LD IX,41298	
3EC0	LD A,192	Check for
CD94A0	CALL CHECK	betinget RET
CA0EA1	JP Z,RET	
3EC2	LD A,194	Check for
CD94A0	CALL CHECK	betinget JP
CACAA0	JP Z,JP	
3EC4	LD A,196	Check for
CD94A0	CALL CHECK	betinget CALL
CADDA0	JP Z,CALL	
0604	LD B,4	Check for
3E20	LD A,32	betinget JR
CD96A0	CALL CHECK + 2	
CAA3A0	JP Z,JR	

FE10	CP 16	Check for DJNZ
CAC0A0	JP Z,DJNZ	
FE18	CP 24	Check for JR
2857	JR Z,E4	
FEC3	CP 195	Check for JP
CAD6A0	JP Z,E1	
FECD	CP 205	Check for CALL
CAEAA0	JP Z,E2 + 1	
FEC9	CP 201	Check for RET
CA1CA1	JP Z,E3	
FEE9	CP 233	Check for
280E	JR Z,E5	JP (HL) og
FEFD	CP 253	JP (IX) og
2805	JR Z,5	JP (IY)
FEDD	CP 221	
C23FA1	JP NZ,EXECUTE	
DD7E01	LD A,(IX + 1)	
18EE	JR − 18	
E5 − 41080		
DD7E00	LD A,(IX + 0)	
FEDD	CP 221	
2005	JR NZ,5	
2A88A4	LD HL,(− IX −)	
180C	JR 12	
FEFD	CP 253	
2005	JR NZ,5	
2A8AA4	LD HL, (− IY −)	
1803	JR 3	
2A86A4	LD HL,(− HL −)	
2290A4	LD (− PC −),HL	
C9	RET	
CHECK − 41108		
0608	LD B,8	Underrutine til
DDBE00	CP (IX + 0)	brug ved test af
C8	RET Z	betingede
C608	ADD A,8	instruktioner
10F8	DJNZ − 8	
DD7E00	LD A,(IX + 0)	
05	DEC B	

```
C9              RET
JR – 41123
32ABA0          LD (41131),A
2A80A4          LD HL,( – AF – )
E5              PUSH HL
F1              POP AF
0001            JR bet. , 1          (adresse 41131)
C9              RET
E4 – 41134
0600            LD B,0
DD4E01          LD C,(IX + 1)
CB79            BIT 7,C
2801            JR Z,1
05              DEC B
2A90A4          LD HL,(– PC – )
09              ADD HL,BC
2290A4          LD (– PC –), HL
C9              RET
DJNZ – 41152
2A82A4          LD HL,( – BC – )
25              DEC H
2282A4          LD ( – BC – ),HL
C8              RET Z
18E4            JR E4
JP – 41162
32D2A0          LD (41170),A
2A80A4          LD HL,( – AF – )
E5              PUSH HL
F1              POP AF
00D6A0          JP bet. , E1         (adresse 41170)
C9              RET
E1 – 41174
2A53A1          LD HL,(41299)        (2. byte i instr.)
2290A4          LD (– PC –),HL
C9              RET
CALL – 41181
32E5A0          LD (41189),A
2A80A4          LD HL,(– AF –)
E5              PUSH HL
F1              POP AF
```

```
00E9A0      CALL bet. , E2          (adresse 41189)
C9          RET
E2 – 41193
E1          POP HL
DD7E02      LD A,(IX + 2)
FEB9        CP 185
D23FA1      JP NC,EXECUTE
ED738EA4    LD (– REAL SP –),SP
2A53A1      LD HL,(41299)           (2. byte i instr.)
ED4B90A4    LD BC,(– PC –)
2290A4      LD (– PC –),HL
ED7B8CA4    LD SP,(– SP –)
C5          PUSH BC
ED738CA4    LD (– SP –),SP
ED7B8EA4    LD SP,(– REAL SP –)
C9          RET
RET – 41230
C602        ADD A,2
3218A1      LD (41240),A
2A80A4      LD HL,(– AF –)
E5          PUSH HL
F1          POP AF
001CA1      JP bet. , E3            (adresse 41240)
C9          RET
E3 – 41244
2A8CA4      LD HL,(– SP –)
0128A0      LD BC,START SP
A7          AND A
ED42        SBC HL,BC
7C          LD A,H
B5          OR L
ED738EA4    LD (– REAL SP –),SP
33          INC SP
33          INC SP
C8          RET Z
ED7B8CA4    LD SP,(– SP –)
E1          POP HL
ED738CA4    LD (– SP –),SP
2290A4      LD (– PC –),HL
```

```
ED7B8EA4     LD SP,(– REAL SP –)
C9           RET
EXECUTE – 41279
ED738EA4     LD (– REAL SP –),SP
3180A4       LD SP,A480h          Bruger-
F1           POP AF               registrene
C1           POP BC               kaldes frem
D1           POP DE               inden
E1           POP HL               eksekvering
DDE1         POP IX
FDE1         POP IY
ED7B8CA4     LD SP,(– SP –)
00000000     DEFB – instruktion –  Udførelse
ED738CA4     LD (– SP –),SP       af instruktion
318CA4       LD SP,A48Ch          Bruger-
FDE5         PUSH IY              registrene
DDE5         PUSH IX              tilbagelægges
E5           PUSH HL
D5           PUSH DE
C5           PUSH BC
F5           PUSH AF
ED7B8EA4     LD SP,(– REAL SP –)
C9           RET
```

```
UDSKRIFT – 41322
3E02         LD A,2
CD90BB       CALL TXT SET PEN
210203       LD HL,770
11A0A3       LD DE,41888          (adressen på
0611         LD B,17              datablok)
CD38A3       CALL PRINT           Udskrivning af
210603       LD HL,774            tekst
0624         LD B,36
CD38A3       CALL PRINT
211217       LD HL,5906
060D         LD B,13
CD38A3       CALL PRINT
3E03         LD A,3
CD90BB       CALL TXT SET PEN
```

```
211417      LD HL,5908
060F        LD B,15
CD38A3      CALL PRINT
3E02        LD A,2
CD90BB      CALL TXT SET PEN
0E07        LD C,7
210804      LD HL,1032
E5          PUSH HL
0601        LD B,1
CD38A3      CALL PRINT
3E3A        LD A,58
CD5ABB      CALL TXT OUTPUT
E1          POP HL
2C          INC L
0D          DEC C
20F0        JR NZ,- 16
0E06        LD C,6
211203      LD HL,786
11F2A3      LD DE,41970
E5          PUSH HL
0602        LD B,2
CD38A3      CALL PRINT
3E3A        LD A,58
CD5ABB      CALL TXT OUTPUT
E1          POP HL
2C          INC L
0D          DEC C
20F0        JR NZ,- 16
C9          RET
```

TALLENE – 41418

```
3E01        LD A,1
CD90BB      CALL TXT SET PEN
210806      LD HL,1544
2292A4      LD (42130),HL
DD2181A4    LD IX,42113
CD82A2      CALL UDSKR
0603        LD B,3
C5          PUSH BC
DD23        INC IX
```

Udskrivning af 8 bits-registre

```
DD23        INC IX
CD82A2      CALL UDSKR
DD2B        DEC IX
CD82A2      CALL UDSKR
DD23        INC IX
C1          POP BC
10EE        DJNZ - 18
DD2182A4    LD IX,42114      Udskrivning af
211207      LD HL,1810       16 bits-registre
0606        LD B,6
C5          PUSH BC
E5          PUSH HL
CD75BB      CALL TXT SET CURSOR
DD6E00      LD L,(IX + 0)
DD6601      LD H,(IX + 1)
E5          PUSH HL
CD43A3      CALL DEC
E1          POP HL
E3          EX (SP),HL
260F        LD H,15
D1          POP DE
E5          PUSH HL
CD75BB      CALL TXT SET CURSOR
7A          LD A,D
CD88A3      CALL HEX
7B          LD A,E
CD88A3      CALL HEX
E1          POP HL
2607        LD H,7
2C          INC L
DD23        INC IX
DD23        INC IX
C1          POP BC
10D5        DJNZ - 43
211617      LD HL,5910       Udskrivning af
CD75BB      CALL TXT SET CURSOR  flagene
3A80A4      LD A,(- AF -)
0608        LD B,8
07          RLCA
```

F5	PUSH AF	
3E30	LD A,48	
CE00	ADC A,0	
CD5ABB	CALL TXT OUTPUT	
3E20	LD A,32	
CD5ABB	CALL TXT OUTPUT	
F1	POP AF	
10EF	DJNZ – 17	
3E03	LD A,3	Udskrivning af
CD90BB	CALL TXT SET PEN	stakken
AF	XOR A	
110D26	LD DE,9741	
210722	LD HL,8711	
CD44BC	CALL SCR FILL BOX	
DD2126A0	LD IX,40998	
2128A0	LD HL,41000	
ED5B8CA4	LD DE,(– SP –)	
A7	AND A	
ED52	SBC HL,DE	
C394A4	JP 42132	
FE07	CP 7	
3802	JR C,2	
3E07	LD A,7	
47	LD B,A	
210823	LD HL,8968	
C5	PUSH BC	
E5	PUSH HL	
CD75BB	CALL TXT SET CURSOR	
DD6E00	LD L,(IX + 0)	
DD6601	LD H,(IX + 1)	
CD43A3	CALL DEC	
DD2B	DEC IX	
DD2B	DEC IX	
E1	POP HL	
2C	INC L	
C1	POP BC	
10E9	DJNZ – 23	
C9	RET	

UDSKR – 41602

2A92A4	LD HL,(42130)	Underrutine til
E5	PUSH HL	brug ved ud-
CD75BB	CALL TXT SET CURSOR	skrift af 8 bits-
DD6E00	LD L,(IX + 0)	registrene i alle
2600	LD H,0	notationer
CD43A3	CALL DEC	
E1	POP HL	
2610	LD H,16	
E5	PUSH HL	
CD75BB	CALL TXT SET CURSOR	
DD7E00	LD A,(IX + 0)	
CD88A3	CALL HEX	
E1	POP HL	
2617	LD H,23	
CD75BB	CALL TXT SET CURSOR	
DD7E00	LD A,(IX + 0)	
CD79A3	CALL BIN	
2A92A4	LD HL,(42130)	
2C	INC L	
2292A4	LD (42130),HL	
C9	RET	

NXT-INS – 41650

210403	LD HL,772	Underrutine, der
CD75BB	CALL TXT SET CURSOR	finder frem til
3E03	LD A,3	næste
CD90BB	CALL TXT SET PEN	instruktions
2152A1	LD HL,41298	hex-kode og
E5	PUSH HL	gemmer denne
0604	LD B,4	på de 4
3600	LD (HL),0	reserverede
23	INC HL	bytes (41298 er
10FB	DJNZ – 5	1. byte i instr.)
DDE1	POP IX	
2A90A4	LD HL,(– PC –)	
E5	PUSH HL	
CD43A3	CALL DEC	
3E20	LD A,32	

```
CD5ABB      CALL TXT OUTPUT
E1          POP HL
0E00        LD C,0
BYTE
7E          LD A,(HL)
FEDD        CP 221
2818        JR Z,DDFD
FEFD        CP 253
2814        JR Z,DDFD
FEED        CP 237
2016        JR NZ, NORMAL
ED
CD19A3      CALL INSTR
7E          LD A,(HL)
E6C7        AND 199
FE43        CP 67
3E01        LD A,1
2002        JR NZ,2
C602        ADD A,2
1819        JR UDSKRIV
DDFD
CD19A3      CALL INSTR
0C          INC C
18DD        JR BYTE
NORMAL
CB3F        SRL A
5F          LD E,A
16A4        LD D,164        (MSB af
1A          LD A,(DE)       datablok)
3804        JR C,LIGE
1F          RRA
1F          RRA
1F          RRA
1F          RRA
LIGE
0D          DEC C
2002        JR NZ,OK
1F          RRA
1F          RRA
```

OK

E603	AND 3

UDSKRIV

47	LD B,A

NÆSTE BYTE

CD19A3	CALL INSTR
10FB	DJNZ NÆSTE BYTE
2290A4	LD (– PC –),HL
C9	RET

INSTR – 41753

7E	LD A,(HL)
DD7700	LD (IX + 0),A
DD23	INC IX
07	RLCA
07	RLCA
07	RLCA
07	RLCA
1602	LD D,2
E60F	AND 15
FE0A	CP 10
3802	JR C,2
C607	ADD A,7
C630	ADD A,48
CD5ABB	CALL TXT OUTPUT
7E	LD A,(HL)
15	DEC D
20EF	JR NZ,– 17
23	INC HL
C9	RET

PRINT – 41784

CD75BB	CALL TXT SET CURSOR
1A	LD A,(DE)
CD5ABB	CALL TXT OUTPUT
13	INC DE
10F9	DJNZ – 7
C9	RET

DEC – 41795		
1E30	LD E,48	Decimal
011027	LD BC,10000	udskrivning
CD5FA3	CALL UNDR	
01E803	LD BC,1000	
CD5FA3	CALL UNDR	
016400	LD BC,100	
CD5FA3	CALL UNDR	
0E0A	LD C,10	
CD5FA3	CALL UNDR	
0E01	LD C,1	
1C	INC E	
UNDR		
3E2F	LD A,47	
3C	INC A	
A7	AND A	
ED42	SBC HL,BC	
30FA	JR NC, – 6	
FE30	CP 48	
2007	JR NZ,7	
BB	CP E	
2004	JR NZ,4	
3E20	LD A,32	
1801	JR 1	
1C	INC E	
CD5ABB	CALL TXT OUTPUT	
09	ADD HL,BC	
C9	RET	
00	NOP	(Ekstra byte)

BIN – 41849		
0608	LD B,8	Binær
07	RLCA	udskrivning
F5	PUSH AF	
3E30	LD A,48	
CE00	ADC A,0	
CD5ABB	CALL TXT OUTPUT	
F1	POP AF	
10F4	DJNZ – 12	
C9	RET	

HEX – 41864

57	LD D,A	Hexadecimal
07	RLCA	udskrivning
07	RLCA	
07	RLCA	
07	RLCA	
0602	LD B,2	
E60F	AND 15	
FE0A	CP 10	
3802	JR C,2	
C607	ADD A,7	
C630	ADD A,48	
CD5ABB	CALL TXT OUTPUT	
7A	LD A,D	
10F0	DJNZ – 16	
C9	RET	

DATA1 – 41888

4144522020	DEFB	Datablok til
20494E5354	DEFB	udskrift
52554B5449	DEFB	
4F4E524547	DEFB	
2020444543	DEFB	
2020202020	DEFB	
4845582020	DEFB	
202042494E	DEFB	
2020202020	DEFB	
2020202053	DEFB	
54414B464C	DEFB	
4147524547	DEFB	
4953545245	DEFB	
5453205A20	DEFB	
9020902090	DEFB	
502F569020	DEFB	
4341424344	DEFB	
45484C4958	DEFB	
49595350	DEFB	

DATA2 – 41982

```
0000        DEFB
5F5555A5    DEFB
555555A5    DEFB
AF5555A5    DEFB
A55555A5    DEFB
AFF555A5    DEFB
A5F555A5    DEFB
AFF599E5    DEFB
A5F555A5    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
99999959    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55555595    DEFB
55FFF5A5    DEFB
55FEFFA5    DEFB
55FAF5A5    DEFB
55FAF5A5    DEFB
55F5F5A5    DEFB
55F5FAA5    DEFB
55F5F5A5    DEFB
55F5F5A5    DEFB
0000        DEFB – AF –
0000        DEFB – BC –
0000        DEFB – DE –
0000        DEFB – HL –
0000        DEFB – IX –
```

Reserveret til den nuværende cursorposition i det testede program.

Datablok til at finde en instruktions-længde

Bruger-registrene

0000	DEFB – IY –	
0000	DEFB – SP –	
0000	DEFB – REAL SP –	
0000	DEFB – PC –	
0000	DEFB – LOCATE-COUNT –	

DELR – 42132

7D	LD A,L	
CB3F	SRL A	
FE00	CP 0	
C8	RET Z	
C360A2	JP 41568	

RETURN – 42141

211801	LD HL,280	Sletter key
CD75BB	CALL TXT SET CURSOR	buffer og sætter
C303BB	JP KM RESET	cursor placering ved slutning

INITIALISE – 42150

019DA4	LD BC,42141	SINGLESTEP-
C5	PUSH BC	PROGRAM-
DD6E00	LD L,(IX+0)	MET's
DD6601	LD H,(IX+1)	startadresse
2290A4	LD (– PC –),HL	Initialisering af
2128A0	LD HL,41000	skærmens
228CA4	LD (– SP –),HL	MODE samt af
CDFFBB	CALL SCR INITIALISE	registrene
2180A4	LD HL,42112	
060C	LD B,12	
3600	LD (HL),0	
23	INC HL	
10FB	DJNZ – 5	
CD6AA1	CALL UDSKRIFT	
210101	LD HL,257	
22FEA3	LD (41982),HL	

DRIVER – 42191

CDCAA1	CALL TALLENE	Hovedrutinen,
3E42	LD A,66	der styrer
CD1EBB	CALL KM TEST KEY	programmet
C0	RET NZ	
3E12	LD A,18	
CD1EBB	CALL KM TEST KEY	
28F3	JR Z, – 13	
AF	XOR A	
210302	LD HL,515	
11030F	LD DE,3843	
CD44BC	CALL SCR FILL BOX	
CDB2A2	CALL NXT – INS	
2AFEA3	LD HL,(41982)	
CD75BB	CALL TXT SET CURSOR	
CD28A0	CALL SINGLESTEP	
CD78BB	CALL TXT GET CURSOR	
22FEA3	LD (41982),HL	
18D2	JR DRIVER	

Når du har indtastet hele programmet, skriv

```
A = 0 : FOR N = 41000 TO 42236 : A = A + PEEK(N) : NEXT :
PRINT A
```

Hvis resultatet ikke bliver 142004, har du lavet en fejl i indtastningen, og du må gennemgå det hele igen.

Gem programmet på bånd inden afprøvning! Læg i øvrigt mærke til, at programmet ikke kun kan bruges til fejlfinding. Vi kan også afprøve de enkelte instruktioners effekt. Det er netop dette, som det følgende vil handle om. Vi vil se, hvordan vores program, som udskriver hex-tal, ser ud. Vores SINGLESTEP-program stiller ikke krav til programmets startadresse, så indtast blot dette fra adresse 40000:

Hex-kode	**Mnemonic**
21021E	LD HL,7682
CD75BB	CALL TXT SET CURSOR
3EC9	LD A,201

57	LD D,A
07	RLCA
07	RLCA
07	RLCA
07	RLCA
0602	LD B,2
E60F	AND 15
FE0A	CP 10
3802	JR C,2
C607	ADD A,7
C630	ADD A,48
CD5ABB	CALL TXT OUTPUT
7A	LD A,D
10F0	DJNZ – 16
C9	RET

Efter indtastningen skriver du

```
CALL 42150,40000
```

Læg mærke til parameteren 40000. Det er her programmets startadresse, og SINGLESTEP-programmet bruger IX-registret på samme måde, som vi gjorde, da vi udvidede BASIC i forrige kapitel.

På skærmen vil du nu kunne se, at alle registre er nulstillet, dog er SP 41000. Tryk nu på <ENTER> (den store af dem). Øverst vil du se, at instruktionen står på adresse 40000, og at det drejer sig om instruktionen med hex-koden 21021E (LD HL,7682). Det var jo også vores første instruktion. Læg også mærke til registrenes ændringer. Tryk <ENTER> igen, og du vil se, at HL og A ændrede sig. Det skyldes selvfølgelig ROM-kaldet. A og HL er altså »ødelagt« af ROM-rutinen, og det vil du også kunne se i kapitel 21. PRINT-positionen er nu sat til 30,2. Prøv nu at køre hele programmet igennem og forstå, hvad der sker. A LOAD'es med 201, så det skulle gerne ende med, at der står C9 (C9h = 201) øverst til højre på skærmen. Tryk <ENTER>, indtil programmet returnerer til BASIC.

Prøv nu selv at afprøve nogle programmer med F.eks. PUSH og POP, så du kan se, hvad der foregår på stakken. Vi skal inden

afslutningen bringe en advarsel. Hvis SP kommer til at indeholde et tal mellem 41000 og 42236 (her ligger SINGLESTEP-programmet), må du under ingen omstændigheder PUSH'e. Det vil jo medføre, at du ændrer i programmet på 2 adresser. Det vil have katastrofale følger.

Hvis du vil ud af programmet, kan du altid trykke på ESC-tasten.

Hermed er vi færdige med bogen – der resterer kun diverse appendices. Du bør nu, hvis du har gennemarbejdet vore eksempler og forklaringer, være udstyret med et solidt fundament til maskinkodeprogrammeringen. Nu er det op til dig selv at videreudvikle din kunnen. Husk i den forbindelse vores demonstrationsprogram fra kapitel 1, som vi ikke har forklaret. Nu kan du selv prøve at forbedre Singlestep-programmet. Det er ikke særligt tilfredsstillende, at programmet kun udskriver instruktionens hex-kode – så skal du hele tiden finde ud af, hvad det er for en instruktion, det drejer sig om. Lav derfor programmet om, så det udskriver *mnemonicen* i stedet. Der er i hvert fald en god træning, da en sådan udvidelse (en *disassembler*) vil fylde (lidt ??) over 1 Kbytes. Men det bliver svært ... Det er op til dig. Fortsat god fornøjelse med maskinkoden!

Appendiks A

I dette appendiks kan du finde en fuldstændig liste over mnemonics og deres tilhørende hex-koder.

Der er benyttet følgende notationer:

n angiver et 8 bits-tal
nn angiver et 16 bits-tal
d angiver et tal to-komplementært

Normale instruktioner

	0	1	1	3
0	NOP	LD BC,nn	LD (BC),A	INC BC
1	DJNZ d	LD DE,nn	LD (DE),A	INC DE
2	JR NZ,d	LD HL,nn	LD (nn),HL	INCX HL
3	JR NC,d	LD SP,nn	LD (nn),A	INC SP
4	LD B,B	LD B,C	LD B,D	LD D,E
5	LD D,B	LD D,C	LD D,D	LD D,E
6	LD H,B	LD H,C	LD H,D	LD H,E
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E
9	SUB B	SUB C	SUB D	SUB E
A	AND B	AND C	AND D	AND E
B	OR B	OR C	OR D	OR E
C	RET NZ	POP BC	JP NZ,nn	JP nn
D	RET NC	POP DE	JP NC,nn	OUT (n),A
E	RET PO	POP HL	JP PO,nn	EX (SP),HL
F	RET P	POP AF	JP P,nn	DI

	4	5	6	7
0	INC B	DEC B	LD B,n	RLCA
1	INC D	DEC D	LD D,n	RLA
2	INC H	DEC H	LD H,n	DAA
3	INC (HL)	DEC (HL)	LD (HL),n	SCF
4	LD B,H	LD B,L	LD B,(HL)	LD B,A
5	LD D,H	LD D,L	LD D,(HL)	LD D,A
6	LD H,H	LD H,L	LD H,(HL)	LD H,A
7	LD (HL),H	LD (HL),L	HALT	LD (HL),A
8	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A
9	SUB H	SUB L	SUB (HL)	SUB A
A	AND H	AND L	AND (HL)	AND A
B	OR H	OR L	OR (HL)	OR A
C	CALL NZ,nn	PUSH BC	ADD A,n	RST 00h
D	CALL NC,nn	PUSH DE	SUB n	RST 10h
E	CALL PO,nn	PUSH HL	AND n	RST 20h
F	CALL P,nn	PUSH AF	OR n	RST 30h

	8	9	A	B
0	EX AF,AF'	ADD HL,BC	LD A,(BC)	DEC BC
1	JR d	ADD HL,DE	LD A,(DE)	DEC DE
2	JR Z,d	ADD HL,HL	LD HL,(nn)	DEC HL
3	JR C,d	ADD HL,SP	LD A,(nn)	DEC SP
4	LD C,B	LD C,C	LD C,D	LD C,E
5	LD E,B	LD E,C	LD E,D	LD E,E
6	LD L,B	LD L,C	LD L,D	LD L,E
7	LD A,B	LD A,C	LD A,D	LD A,E
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E
A	XOR B	XOR C	XOR D	XOR E
B	CPB	CPC	CPD	CPE
C	RET Z	RET	JP Z,nn	●
D	RET C	EXX	JP C,nn	IN A,(n)
E	RET PE	JP (HL)	JP PE,nn	EX DE,HL
F	RET M	LD SP,HL	JP M,nn	EI

	C	D	E	F
0	INC C	DEC C	LD C,n	RRCA
1	INC E	DEC E	LD E,n	RRA
2	INC L	DEC L	LD L,n	CPL
3	INC A	DEC A	LD A,n	CCF
4	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,H	LD L,L	LD L,(HL)	LD A,A
7	LD A,H	LD A,L	LD A,(HL)	ADC A,A
8	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR H	XOR L	XOR (HL)	XOR A
B	CP H	CP L	CP (HL)	CP A
C	CALL Z,nn	CALL nn	ADC A,n	RST 08h
D	CALL C,nn	●	SBC A,n	RST 18h
E	CALL PE,nn	●	XOR n	RST 28h
F	CALL M,nn	●	CP n	RST 38h

Efter CB:

	0	1	2	3	4	5	6	7
0	RLC B	RLC C	RLC D	RLC E	RLCH	RLC L	RLC (HL)	RLC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A
3	–	–	–	–	–	–	–	–
4	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A
8	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A
C	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A

	8	9	A	B	C	D	E	F
0	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

Efter DD:

	0	1	2	3	4	5	6
0	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–
2	–	LD IX,nn	LD (nn),IX	INC IX	–	–	–
3	–	–	–	–	INC (IX + d)	DEC (IX + d)	LD (IX + d),n
4	–	–	–	–	–	–	LD B,(IX + d)
5	–	–	–	–	–	–	LD D,(IX + d)
6	–	–	–	–	–	–	LD H,(IX + d)
7	LD (IX + d),B	LD (IX + d),C	LD (IX + d),D	LD (IX + d),E	LD (IX + d),H	LD (IX + d),L	–
8	–	–	–	–	–	–	ADD A,(IX + d)
9	–	–	–	–	–	–	SUB (IX + d)
A	–	–	–	–	–	–	AND (IX + d)
B	–	–	–	–	–	–	OR (IX + d)
C	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–
E	–	POP IX	–	EX (SP),IX	–	PUSH IX	–
F	–	–	–	–	–	–	–

	7	8	9	A	B	C	D	E	F
0	–	–	ADD IX,BC	–	–	–	–	–	–
1	–	–	ADD IX,DE	–	–	–	–	–	–
2	–	–	ADD IX,IX	LD IX,(nn)	DEC IX	–	–	–	–
3	–	–	ADD IX,SP	–	–	–	–	–	–
4	–	–	–	–	–	–	–	LD C,(IX + d)	–
5	–	–	–	–	–	–	–	LD E,(IX + d)	–
6	–	–	–	–	–	–	–	LD A,(IX + d)	–
7	LD (IX + d),A	–	–	–	–	–	–	LD A,(IX + d)	–
8	–	–	–	–	–	–	–	ADC A,(IX + d)	–
9	–	–	–	–	–	–	–	SBC A,(IX + d)	–
A	–	–	–	–	–	–	–	XOR (IX + d)	–
B	–	–	–	–	–	–	–	CP (IX + d)	–
C	–	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–	–
E	–	–	JP (IX)	–	EX DE,IX	–	–	–	–
F	–	–	LD SP,IX	–	–	–	–	–	–

Efter ED:

	0	1	2	3	4	5	6	7
0	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–
4	IN B,(C)	OUT (C),B	SBC HL,BC	LD (nn),BC	NEG	RETN	IM 0	LD I,A
5	IN D,(C)	OUT (C),D	SBC HL,DE	LD (nn),DE	–	–	IM 1	LD A,I
6	IN H,(C)	OUT (C),H	SBC HL,HL	LD (nn),HL	–	–	–	RRD
7	–	–	SBC HL,SP	LD (nn),SP	–	–	–	–
8	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–
A	LDI	CPI	INI	OUTI	–	–	–	–
B	LDIR	CPIR	INIR	OTIR	–	–	–	–
C	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–
E	–	–	–	–	–	–	–	–
F	–	–	–	–	–	–	–	–

	8	9	A	B	C	D	E	F
0	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–
4	IN C,(C)	OUT (C),C	ADC HL,BC	LD BC,(nn)	–	RETI	–	LD R,A
5	IN E,(C)	OUT (C),E	ADC HL,DE	LD DE,(nn)	–	–	IM 2	LD A,R
6	IN L,(C)	OUT (C),L	ADC HL,HL	LD HL,(nn)	–	–	–	RLD
7	IN A,(C)	OUT (C),A	ADC HL, SP	LD SP,(nn)	–	–	–	–
8	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–
A	LDD	CPD	IND	OUTD	–	–	–	–
B	LDDR	CPDR	INDR	OTDR	–	–	–	–
C	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–
E	–	–	–	–	–	–	–	–
F	–	–	–	–	–	–	–	–

Efter FD:

	0	1	2	3	4	5	6
0	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–
2	–	LD IY,nn	LD (nn),IY	INC IY	–	–	–
3	–	–	–	–	INC (IY+d)	DEC (IY + d)	LD (IY + d),n
4	–	–	–	–	–	–	LD B,(IY + d)
5	–	–	–	–	–	–	LD D,(IY + d)
6	–	–	–	–	–	–	LD H,(IY + d)
7	LD (IY + d),B	LD (IY + d),C	LD (IY + d),D	LD(IY + d),E	LD (IY + d),H	LD (IY + d),L	–
8	–	–	–	–	–	–	ADD A,(IY + d)
9	–	–	–	–	–	–	SUB (IY + d)
A	–	–	–	–	–	–	AND (IY + d)
B	–	–	–	–	–	–	OR (IY + d)
C	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–
E	–	POP IY	–	EX (SP),IY	–	PUSH IY	–
F	–	–	–	–	–	–	–

	7	8	9	A	B	C	D	E	F
0	–	–	ADD IY,BC	–	–	–	–	–	–
1	–	–	ADD IY,DE	–	–	–	–	–	–
2	–	–	ADD IY,IY	LD IY,(nn)	DEC IY	–	–	–	–
3	–	–	ADd IY,SP	–	–	–	–	–	–
4	–	–	–	–	–	–	–	LD C,(IY + d)	–
5	–	–	–	–	–	–	–	LD E,(IY + d)	–
6	–	–	–	–	–	–	–	LD L,(IY + d)	–
7	LD (IY + d),A	–	–	–	–	–	–	LD A,(IY + d)	–
8	–	–	–	–	–	–	–	ADC A,(IY + d)	–
9	–	–	–	–	–	–	–	SBC A,(IY + d)	–
A	–	–	–	–	–	–	–	XOR (IY + d)	–
B	–	–	–	–	–	–	–	CP (IY + d)	–
C	–	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–	–
E	–	–	JP (IY)	–	EX DE,IY	–	–	–	–
F	–	–	LD SP,IY	–	–	–	–	–	

Efter DDCB:

	6	E
0	RLC (IX + d)	RRC (IX + d)
1	RL (IX + d)	RR (IX + d)
2	SLA (IX + d)	SRA (IX + d)
3	–	SRL (IX + d)
4	BIT 0,(IX + d)	BIT 1,(IX + d)
5	BIT 2,(IX + d)	BIT 3,(IX + d)
6	BIT 4,(IX + d)	BIT 5,(IX + d)
7	BIT 6,(IX + d)	BIT 7,(IX + d)
8	RES 0,(IX + d)	RES 1,(IX + d)
9	RES 2,(IX + d)	RES 3,(IX + d)
A	RES 4,(IX + d)	RES 5,(IX + d)
B	RES 6,(IX + d)	RES 7,(IX + d)
C	SET 0,(IX + d)	SET 1,(IX + d)
D	SET 2,(IX + d)	SET 3,(IX + d)
E	SET 4,(IX + d)	SET 5,(IX + d)
F	SET 6,(IX + d)	SET 7,(IX + d)

Efter FDCB:

	6	E
0	RLC (IY + d)	RRC (IY + d)
1	RL (IY + d)	RR (IY + d)
2	SLA (IY + d)	SRA (IY + d)
3	–	SRL (IY + d)
4	BIT 0,(IY + d)	BIT 1,(IY + d)
5	BIT 2,(IY + d)	BIT 3,(IY + d)
6	BIT 4,(IY + d)	BIT 5,(IY + d)
7	BIT 6,(IY + d)	BIT 7,(IY + d)
8	RES 0,(IY + d)	RES 1,(IY + d)
9	RES 2,(IY + d)	RES 3,(IY + d)
A	RES 4,(IY + d)	RES 5,(IY + d)
B	RES 6,(IY + d)	RES 7,(IY + d)
C	SET 0,(IY + d)	SET 1,(IY + d)
D	SET 2,(IY + d)	SET 3,(IY + d)
E	SET 4,(IY + d)	SET 5,(IY + d)
F	SET 6,(IY + d)	SET 7,(IY + d)

Appendiks B

I dette appendiks kigger vi på flagsætning. Vi bruger følgende notationer:

1:	flaget sættes
0:	flaget resættes
*:	flaget uberørt
↓:	flaget ændres afhængigt af resultatet
P:	P/V-flaget viser partiet
V:	P/V-flaget indikerer overflow
X:	flagets værdi ukendt
S:	S-flaget (Sign-flaget)
Z:	Z-flaget (Zero-flaget)
P/V:	P/V-flaget (Partiet/Overflow-flaget)
C:	C-flaget (Carry-flaget)

	S	Z	P/V	C
8 bits-ADC	↓	↓	V	↓
8 bits-ADD	↓	↓	V	↓
16 bits-ADD	*	*	*	↓
AND	↓	↓	P	0
BIT	X	↓	X	*
CALL	*	*	*	*
CCF	*	*	*	↓
CP	↓	↓	V	↓
CPD	↓	↓	↓	*
CPDR	↓	↓	↓	*
CPI	↓	↓	↓	*
CPIR	↓	↓	↓	*
CPL	*	*	*	*
DAA	↓	↓	P	↓
8 bits-DEC	↓	↓	V	*
16 bits-DEC	*	*	*	*
DI	*	*	*	*
DJNZ	*	*	*	*
EI	*	*	*	*
EX	*	*	*	*
EXX	*	*	*	*
HALT	*	*	*	*
IM	*	*	*	*
IN A,n	*	*	*	*
IN r,(C)	↓	↓	P	*
8 bits-INC	↓	↓	V	*
16 bits-INC	*	*	*	*
IND	X	↓	X	*
INDR	X	1	X	*
INI	X	↓	X	*
INIR	X	1	X	*

	S	Z	P/V	C
JP	*	*	*	*
JR	*	*	*	*
LD_1	*	*	*	*
LDD	*	*	↓	*
LDDR	*	*	0	*
LDI	*	*	↓	*
LDIR	*	*	0	*
NEG	↓	↓	V	↓
NOP	*	*	*	*
OR	↓	↓	P	0
OUTI	X	↓	X	*
OTIR	X	1	X	*
OUT	*	*	*	*
OUTD	X	↓	X	*
OTDR	X	1	X	*
POP_2	*	*	*	*
PUSH	*	*	*	*
RES	*	*	*	*
RET	*	*	*	*
RETI	*	*	*	*
RETN	*	*	*	*
RL	↓	↓	P	↓
RLA	*	*	*	↓
RLC	↓	↓	P	↓
RLCA	*	*	*	↓
RLD	↓	↓	P	*
RR	↓	↓	P	↓
RRA	*	*	*	↓
RRC	↓	↓	P	↓
RRCA	*	*	*	↓
RRD	↓	↓	P	*
RST	*	*	*	*
8-bits SBC	↓	↓	V	↓
16-bits SBC	↓	↓	V	↓
SCF	*	*	*	1
SET	*	*	*	*
SLA	↓	↓	P	↓

	S	Z	P/V	C
SRA	↓	↓	P	↓
SRL	↓	↓	P	↓
SUB	↓	↓	V	↓
XOR	↓	↓	P	0

1) LD A,I og LD A,R påvirker S og Z afhængig af resultatet.
2) POP AF vil selvfølgelig ændre flagene afhængig af tallet på stakken.

Appendiks C

DEC.	HEX.	DEC.	HEX.	DEC.	HEX.	DEC.	HEX.
0	00	32	20	64	40	96	60
1	01	33	21	65	41	97	61
2	02	34	22	66	42	98	62
3	03	35	23	67	43	99	63
4	04	36	24	68	44	100	64
5	05	37	25	69	45	101	65
6	06	38	26	70	46	102	66
7	07	39	27	71	47	103	67
8	08	40	28	72	48	104	68
9	09	41	29	73	49	105	69
10	0A	42	2A	74	4A	106	6A
11	0B	43	2B	75	4B	107	6B
12	0C	44	2C	76	4C	108	6C
13	0D	45	2D	77	4D	109	6D
14	0E	46	2E	78	4E	110	6E
15	0F	47	2F	79	4F	111	6F
16	10	48	30	80	50	112	70
17	11	49	31	81	51	113	71
18	12	50	32	82	52	114	72
19	13	51	33	83	53	115	73
20	14	52	34	84	54	116	74
21	15	53	35	85	55	117	75
22	16	54	36	86	56	118	76
23	17	55	37	87	57	119	77
24	18	56	38	88	58	120	78
25	19	57	39	89	59	121	79
26	1A	58	3A	90	5A	122	7A
27	1B	59	3B	91	5B	123	7B
28	1C	60	3C	92	5C	124	7C
29	1D	61	3D	93	5D	125	7D
30	1E	62	3E	94	5E	126	7E
31	1F	63	3F	95	5F	127	7F

DEC.	HEX.	2-KOMPL.	DEC.	HEX.	2-KOMPL.
128	80	−128	160	A0	−96
129	81	−127	161	A1	−95
130	82	−126	162	A2	−94
131	83	−125	163	A3	−93
132	84	−124	164	A4	−92
133	85	−123	165	A5	−91
134	86	−122	166	A6	−90
135	87	−121	167	A7	−89
136	88	−120	168	A8	−88
137	89	−119	169	A9	−87
138	8A	−118	170	AA	−86
139	8B	−117	171	AB	−85
140	8C	−116	172	AC	−84
141	8D	−115	173	AD	−83
142	8E	−114	174	AE	−82
143	8F	−113	175	AF	−81
144	90	−112	176	B0	−80
145	91	−111	177	B1	−79
146	92	−110	178	B2	−78
147	93	−109	179	B3	−77
148	94	−108	180	B4	−76
149	95	−107	181	B5	−75
150	96	−106	182	B6	−74
151	97	−105	183	B7	−73
152	98	−104	184	B8	−72
153	99	−103	185	B9	−71
154	9A	−102	186	BA	−70
155	9B	−101	187	BB	−69
156	9C	−100	188	BC	−68
157	9D	−99	189	BD	−67
168	9E	−98	190	BE	−66
159	9F	−97	191	BF	−65

DEC.	HEX.	2-KOMPL.	DEC.	HEX.	2-KOMPL.
192	C0	− 64	224	E0	− 32
193	C1	− 63	225	E1	− 31
194	C2	− 62	226	E2	− 30
195	C3	− 61	227	E3	− 29
196	C4	− 60	228	E4	− 28
197	C5	− 59	229	E5	− 27
198	C6	− 58	230	E6	− 26
199	C7	− 57	231	E7	− 25
200	C8	− 56	232	E8	− 24
201	C9	− 55	233	E9	− 23
202	CA	− 54	234	EA	− 22
203	CB	− 53	235	EB	− 21
204	CC	− 52	236	EC	− 20
205	CD	− 51	237	ED	− 19
206	CE	− 50	238	EE	− 18
207	CF	− 49	239	EF	− 17
208	D0	− 48	240	F0	− 16
209	D1	− 47	241	F1	− 15
210	D2	− 46	242	F2	− 14
211	D3	− 45	243	F3	− 13
212	D4	− 44	244	F4	− 12
213	D5	− 43	245	F5	− 11
214	D6	− 42	246	F6	− 10
215	D7	− 41	247	F7	− 9
216	D8	− 40	248	F8	− 8
217	D9	− 39	249	F9	− 7
218	DA	− 38	250	FA	− 6
219	DB	− 37	251	FB	− 5
220	DC	− 36	252	FC	− 4
221	DD	− 35	253	FD	− 3
222	DE	− 34	254	FE	− 2
223	DF	− 33	255	FF	− 1

Appendiks D

I dette appendiks er opgivet udførelsestiden for instruktionerne. Tiden måles i milliontedele af et sekund (mikrosekunder). Udførelsestiden afhænger af to ting; dels af den clockfrekvens, som processoren kører med, og dels af instruktionens art.

I Amstrad er clockfrekvensen opgivet til 4 MHz. Imidlertid er Amstrad opbygget på en sådan måde, at den effektive clockfrekvens kun er ca. 3,3 MHz. At Z80 kører med 3,3 MHz betyder, at den svinger 3,3 millioner gange i sekundet.

Z80 behøver, afhængig af de forskellige instruktioners kompleksitet, mellem 4 og 23 svingninger for at udføre en instruktion. Dette antal svingninger kaldes for clockcycles. Når man ved, hvor mange clockcycles (eller T-states som det også hedder) en instruktion varer, kan man udregne udførelsestiden (i mikrosekunder) med dette udtryk:

$$\text{udførelsestid} = \frac{\text{antal clockcycles}}{\text{clockfrekvens i MHz}}$$

For eksempel varer instruktionen LD A,(BC) 7 clockcycles. I Amstrad tager det derfor 7 : 3,3 = 2,12 mikrosekunder at udføre instruktionen.

Det følgende skema viser de enkelte instruktioners varighed i clockcycles. Hvis der står to forskellige værdier for nogle instruktioner som f.eks. JR NZ,d, gælder den første, hvis betingelsen er opfyldt, og den anden, hvis den ikke er opfyldt. For de repeterende blokinstruktioner gælder den første, hvis den gentages, og den anden hvis den ikke skal.

Der benyttes flg. forkortelser:

n	= 8 bits-tal
nn	= 16 bits-tal
d	= to-komplementært tal (mellem – 128 og 127)
x	= et tal mellem 0 og 7
r	= et af registrene A, B, C, D, E, H, L
rr	= et af registrene BC, DE, HL, SP
INDEX	= et af registrene IX, IY
bet	= betingelse

ADC A,r	4
ADC A,(HL)	7
ADC A,(INDEX + d)	19
ADC A,n	7
ADC HL,rr	15
ADD A,r	4
ADD A,(HL)	7
ADD A,(INDEX + d)	19
ADD A,n	7
ADD HL,rr	11
ADD INDEX,BC	15
ADD INDEX,DE	15
ADD INDEX,INDEX	15
ADD INDEX,SP	15
AND r	4
AND (HL)	7
AND (INDEX + d)	19
AND n	7
BIT x,r	8
BIT x,(HL)	12
BIT x,(INDEX + d)	20
CALL nn	17
CALL bet,nn	17/10
CCF	4
CP r	4
CP (HL)	7
CP (INDEX + d)	19
CP n	7
CPD	16
CPDR	21/16
CPI	16
CPIR	21/16
CPL	4
DAA	4
DEC r	4
DEC (HL)	11
DEC (INDEX + d)	23
DEC rr	6
DEC INDEX	10

DI	4
DJNZ d	13/8
EI	4
EX (SP),HL	19
EX (SP),INDEX	23
EX AF,AF'	4
EX DE,HL	4
EXX	4
HALT	4
IM	8
IN r,(C)	12
IN A,(n)	11
INC r	4
INC (HL)	11
INC (INDEX + d)	23
INC rr	6
INC INDEX	10
IND	16
INDR	21/16
INI	16
INIR	21/16
JP (HL)	4
JP (INDEX)	8
JP nn	10
JP bet,nn	10
JR d	12
JR bet,d	12/7
LD (nn),A	13
LD (nn),rr	20
LD (nn),HL	16
LD (nn),INDEX	20
LD (BC),A	7
LD (DE),A	7
LD (HL),A	7
LD (HL),r	7
LD (HL),n	10
LD (INDEX + d),r	19
LD (INDEX + d),n	19
LD A,(nn)	13

LD A,(BC)	7
LD A,(DE)	7
LD A,(HL)	7
LD A,I	9
LD A,R	9
LD r,r	4
LD r,(HL)	7
LD r,(INDEX + d)	19
LD r,n	7
LD rr,nn	10
LD rr,(nn)	20
LD HL,(nn)	16
LD I,A	9
LD INDEX,nn	14
LD INDEX,(nn)	20
LD R,A	9
LD SP,HL	6
LD SP,INDEX	10
LDD	16
LDDR	21/16
LDI	16
LDIR	21/16
NEG	8
NOP	4
OR r	4
OR (HL)	7
OR (INDEX + d)	19
OR n	7
OTDR	21/16
OTIR	21/16
OUT (C),r	12
OUT (n),A	11
OUTD	16
OUTI	16
POP AF	10
POP BC	10
POP DE	10
POP HL	10
POP INDEX	14

PUSH AF	11
PUSH BC	11
PUSH DE	11
PUSH HL	11
PUSH INDEX	15
RES x,r	8
RES x,(HL)	15
RES x,(INDEX + d)	23
RET	10
RET bet	11/5
RETI	14
RETN	14
RLA	4
RRA	4
RLCA	4
RRCA	4
RL r	8
RLC r	8
RR r	8
RRC r	8
RL (HL)	15
RLC (HL)	15
RR (HL)	15
RRC (HL)	15
RL (INDEX + d)	23
RLC (INDEX + d)	23
RR (INDEX + d)	23
RRC (INDEX + d)	23
RLD	18
RRD	18
RST	11
SBC A,r	4
SBC A,(HL)	7
SBC A,(INDEX + d)	19
SBC A,n	7
SBC HL,rr	15
SCF	4
SET x,r	8
SET x,(HL)	15

SET x,(INDEX + d)	23
SLA r	8
SRA r	8
SRL r	8
SLA (HL)	15
SRA (HL)	15
SRL (HL)	15
SLA (INDEX + d)	23
SRA (INDEX + d)	23
SRL (INDEX + d)	23
SUB r	4
SUB (HL)	7
SUB (INDEX + d)	19
SUB n	7
XOR r	4
XOR (HL)	7
XOR (INDEX + d)	19
XOR n	7

I tabellen står der LD, HL,(nn) to gange – nemlig som LD HL,(nn) og LD rr,(nn). Grunden til dette er, at den sidste af de to er en instruktion startende med EDh, og den varer 20 T-states, mens den »normale« kun bruger 16 T-states. Der findes således 2 LD HL,(nn) med to forskellige hex-koder (2Ann – ED6Bnn), men der er ingen forskel på dem. Det samme gør sig gældende for LD (nn),HL.

Appendix E

Kontrolkoder

Kode	Navn	Parametre	Funktion
0	NUL	0	Ingen
1	SOH	1	Udskriver karakteren svarende til parameteren
2	STX	0	Fjerner markøren
3	ETX	0	Sætter markør
4	EOT	1	Sætter skærmens MODE – parameteren angiver hvilken. Svarer til at kalde SCR SET MODE
5	ENQ	1	Udskriver en karakter givet af parameteren på grafikmarkørens position. Svarer til kald af GRA WR CHAR
6	ACK	0	Tilkobler tekstskærmen
7	BEL	0	Giver en lyd
8	BS	0	Flytter markøren en plads til venstre
9	TAB	0	Flytter markøren en plads til højre
10	LF	0	Flytter markøren en linje ned
11	VT	0	Flytter markøren en linje op
12	FF	0	Rydder det vindue, som der arbejdes i, og flytter markøren til øverste venstre hjørne. Svarer til kald af TXT CLEAR WINDOW

13	CR	0	Flytter markøren helt til venstre i det nuværende vindue
14	SO	1	Sætter baggrundsfarven til det af parameteren angivne
15	SI	1	Sætter pennen til det af parameteren angivne
16	DLE	0	Sletter karakteren på den nuværende markørposition
17	DC1	0	Sletter fra vinduets venstre kant til markørpositionen incl.
18	DC2	0	Sletter fra markørpositionen til vinduets højre kant
19	DC3	0	Sletter fra vinduets øverste venstre hjørne til og med markørpositionen
20	DC4	0	Sletter fra markørpositionen til vinduets nederste højre kant
21	NAK	0	Frakobler tekstskærmen. Denne vil ikke reagere, før ACK (6) er sendt
22	SYN	1	Parameteren angiver »character write mode«. Hvis parameteren er nul, frakobles »transparent mode«, mens en tilkobler den. »Transparent mode« vil sige, at alt, hvad der skrives på skærmen, vil blive oveni det, der står i forvejen. Se desuden TXT SET BACK

23	ETB	1	Sætter »graphic write mode«, der angives af parameteren: 0 = force mode 1 = XOR mode 2 = AND mode 3 = OR mode
24	CAN	0	Ombytter pen- og baggrundsfarven – se TXT INVERSE
25	EM	9	Sætter matricen for en karakter. Den første parameter angiver hvilken karakterkode og de sidste 8 angiver matricen. Hvis karakteren ikke er brugerdefinerbar, sker der intet, se TXT SET MATRIX
26	SUB	4	Sætter vinduets størrelse. De to første parametre specificerer vinduets bredde (angivet i kolonnenumre), mens de 2 sidste specificerer højden (angivet i linjenumre)
27	ESC	0	Ingen – kan benyttes af brugeren
28	FS	3	Sætter en INK. Første parameter angiver INK-nummeret, mens de to næste angiver farverne. Se SCR SET INK

29	GS	2	Parametrene angiver hvilke farver BORDER skal have. Se SCR SET BORDER
30	RS	0	Flytter markørens position til vinduets øverste venstre hjørne
31	US	2	Flytter markøren som angivet. Første parameter angiver kolonnen, anden parameter rækken (linjen). Se TXT SET CURSOR

Stikordsregister

Addition *46*
Adresse *13, 23, 28*
Adressebus *98*
Adresseringsområde *103*
Akkumulator *17*
Alternative registre *172*
AND *37*
Assembler *18*

Baggrund *126, 135*
BASIC-fortolker *11, 103*
BCD-aritmetik *91*
BIT *37*
Bit *12, 86*
Bit-instruktioner *36*
Binære talsystem *11, 90*
Blokinstruktioner *82*
Bloksøgeinstruktioner *61*
Bogstaver *41*
Byte *12*

C-flaget *34, 63, 86*
CALL *19, 74*
Clockfrekvens *350*
CP *58*
CPL *63*
CPU *12*
Cursor *124*

DAA *34, 92*
DEC *43*
Decimale tal *13, 80, 347*
DEFB *28*
Demonstration *14*
DJNZ *56*

Envelope *157*
Etiket *81*
Events *165*
EX *63, 72*
Explode *296*

F-registret; se Flagene
Firmware; se Operativsystem
Flag *136*
Flagene *32, 45, 49, 73, 343*
Fortegnsbit; se Sign bit
Frame flyback interrupt *164, 168*

Grafikskærm *134*

Hex *299*
Hexadecimale tal *12, 96, 347*
Hex-kode *18, 96*
Hex-load'er *14*
High Kernel *273*
HOJRE *300, 305*
Hop *52*

I-registret *32*
IN *98*
INC *43, 71*
Index-registre *24*
Instruktion *18, 182*
Interrupts *112, 163, 171*

Joystick *118*
JP *52*
JR *54*
Jumpblock *106, 115, 187, 273*

Karakter *59, 113, 120, 128, 132, 140*
Karaktergenerator *128*
Kassettebåndoptager *237*
Kernel *257*
Keyboard' se Tastatur
Keybuffer *113*
Kommando *296*
Kontrolkode *59*

LINE *134*
Logiske instruktioner *37*
LROM *103*
LSB *31*
Lyd *156, 250, 296*

Machine Pack *269*
Maskinkode *11*
Matrix *120, 126*
Memory-mapping *103*
Mente *34*
Mnemonic *18*
MODE *122*
MOVE *134*
MSB *31*

NED *301, 305*
NEG *63*
NOP *63*

OP *301, 305*
Operativsystem *103*
OR *39*
Origo *134*
OUT *99*
Overflowindikator *34, 50*

P/V-flaget *34*
Paritetsindikator *34, 39*
PC (Program Counter) *32, 52, 75*
Pen *125, 135*
Pixelscroll *149, 153*
PLOT *134*
POP *67*
Port *98*
Program *106*
PUSH *65*

R-registret *32*
RAM *12, 103*
Registre *17, 32, 115*
RES *36*
Reset *107*
RET *19, 75*
ROM *12, 78, 103*
Rotationsinstruktioner *86*
RST *107*
Rutine *19*
Råd *179*

Sammenligning *58*
Screen *297*
Scroll *127, 149, 300*
SET *36*
Sign bit *33, 50*
Singlestep *182, 310, 313*
Skifteinstruktioner *88*
Skærm *224*
Skærmbillede *83, 106, 120*
SP (Stack Pointer) *32, 65*
Spring *52*
Stak *65, 75, 106*
Stakinstruktioner *71*
Stream *126*
Subrutine *74*
Subtraktion *46*
System clock *165*
Søjlediagram *140*

T-states *350*
Taste *116*
Tastatur *113, 195*
Tekstskærm *120, 204*
Ticker interrupt *164*

To-komplement-metode *26, 33, 50, 347*

Udvidelseskarakterer *114, 115*
Underprogram *74*
Ur *278, 288*
UROM *103*

Variable *106*
Venstre *300, 305*
Vindue *125, 139*
Vuffelivov *160*

XOR *40*

Z80A *12*
Z-flag *33*

Jørn Lorentzen og Henrik Nellagers nye bog handler om maskinkodeprogrammering på de tre Amstrad computere CPC 464, CPC 664 og CPC 6128.

Forfatterne starter helt fra grunden med binære tal og de simpleste maskinkode-ordrer. Derefter gennemgås ordresættet for Z80 – processoren og Amstrad computerens særlig egenskaber med hensyn til grafik, farver og lyd.

Bogen indeholder desuden mere end 90 siders gennemgang af Amstrad computernes firmware-rutiner, der kan benyttes som byggesten i dine egne programmer.

Overalt i bogen er der programeksempler, som du kan taste ind og lære af, bl.a. et single-step program, der gør det muligt at afvikle programmer i ét programskridt ad gangen, så du hele tiden kan se hvad der sker inde i computeren.

Bogen er lige velegnet som håndbog for den erfarne og som kursus for begynderen.

Borgen

ISBN 87-418-7533-8

Jørn Lorentzen og Henrik Nellager **Maskinkode med Amstrad** Borgen